A solution approach to non-linear multi-field problems.

Sustar, Tomaz

DEPARTMENT OF CIVIL ENGINEERING
UNIVERSITY OF WALES SWANSEA

A SOLUTION APPROACH TO NON-LINEAR MULTI-FIELD PROBLEMS

Tomaž Šuštar
Dipl.Ing. Metalurgy
Faculty of Natural Sciences and Engineering
University of Ljubljana

THESIS SUBMITTED TO THE UNIVERSITY OF WALES
IN CANDIDATURE FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DECEMBER 2002

Acknowledgments

I would like to thank:

- my supervisors, Prof. D.R.J. Owen and Dr. Jože Korelc, for their guidance and advice;

- Dr. Tomaž Rodič, for his unconditional support;

- C3M for financial support;

- my mother Sonja.

Summary

In this work a solution approach for non-linear multi-filed problems is presented. The approach is based on co-operative usage of several advanced techniques inside a single environment instead of combining several different systems.

The objective of this work is to demonstrate the applicability of advanced computational techniques to complex numerical problems and to present advantages of a co-operative solution environment in the development of finite elements.

The solution environment, implemented in *Mathematica*, consists of a symbolic code generator - *AceGen*, a package of prearranged modules for the automatic creation of the interfaces between the generated code and specific finite element environment - *Computational Templates* and a model finite element environment called *Finite Element Driver*.

Within the scope of this work the ANSI C version of *Finite Element Driver* - *CDriver* was developed and used for numerical evaluation throughout the work. The *CDriver* is fully integrated with *Mathematica* and it provides high numerical efficiency to the environment.

The solution approach is demonstrated on magneto-thermo-mechanical problem of inductive heat treatment. First the high abstract formulation level, which is required for efficient symbolic description, was introduced. Following the general formulation the models of individual magnetic, thermal and displacement fields were derived. After the individual fields model were verified the magneto-thermal and magneto-thermo-mechanical problems were formulated and derived. Both non-linear multi-filed models were verified using analytical solutions and numerical convergence tests.
Different multi-filed solution strategies were applied to numerical examples and their performance issues were studied using the magneto-thermo-mechanical model.

Finally the large scale numerical example of inductive heat treatment was solved.

Contents

1 INTRODUCTION 5

1.1 MOTIVATION AND SCOPE 5
1.2 PRESENTATION ORDER 7

2 FINITE ELEMENT SOLUTION ENVIRONMENTS 8

2.1 STATE OF THE ART IN GENERATION OF FE CODES 8
2.1.1 SAC – symbolic and algebraic systems 9
2.1.2 Automatic differentiation tools 9
2.1.3 Theorem proving systems 10
2.1.4 Automatic code generators 10
2.2 STATE OF THE ART IN SOLUTION ENVIRONMENTS 10
2.2.1 Object oriented solutions 11
2.2.2 Problem solving environments 12
2.2.3 Hybrid environments 13

3 CO-OPERATIVE APPROACH 16

3.1 ACEGEN SYMBOLIC CODE GENERATOR 18
3.1.1 Simultaneous optimization 18
3.1.2 AceGen characteristic steps 19
3.1.3 Assignment operators 21
3.1.4 Symbolic Input/Output Interface 22
3.1.5 Flow control operators 24
3.1.6 Automatic differentiation 25
3.1.7 AceGen user interface 26
3.2 COMPUTATIONAL TEMPLATES 28
3.2.1 Characteristic example 29
3.2.2 Computational Templates interface commands 35
3.3 FINITE ELEMENT DRIVER 38
3.3.1 Characteristic example of Finite Element Driver usage 39
3.3.2 Finite Element Driver basic data structures 42
3.3.3 Finite Element Driver problem input data 49
3.3.4 Finite Element Driver analysis 54
3.3.5 Finite Element Driver post-processing 55
3.3.6 CDriver implementation details 57

4 IMPLICIT SOLUTION METHODS FOR NON-LINEAR SYSTEMS 66

4.1 GENERAL FORMULATION 66
4.2 STEADY STATE NON-LINEAR SYSTEMS 67
4.3 TRANSIENT NON-LINEAR SYSTEMS 67
4.4 STEADY STATE COUPLED NON-LINEAR SYSTEM 68
4.5 TRANSIENT COUPLED NON-LINEAR SYSTEMS 70

5 THE FINITE ELEMENT METHOD 72

5.1 FINITE ELEMENT FORMULATION 72
5.2 ELEMENT LEVEL DESCRIPTION 74
5.3 ISOPARAMETRIC ELEMENTS 75
5.3.1 L1 – Linear element with two nodes 77

5.3.2 *T1 - Triangle element with three nodes* 77
5.3.3 *Q1 – Quadrilateral with four nodes* 78
5.3.4 *Q2 – Quadrilateral with eight nodes* 79
5.4 NUMERICAL INTEGRATION 81
5.4.1 *One dimensional numerical integration* 81
5.4.2 *Two dimensional numerical integration* 82

6 FORMULATION OF MAGNETIC, THERMAL AND MECHANICAL PROBLEM 86

6.1 MAGNETIC FIELD 87
6.1.1 *Magnetic element implementation* 90
6.1.2 *Verification of the magnetic element* 93
6.2 THERMAL FIELD 97
6.2.1 *Thermal element implementation* 98
6.2.2 *Surface flux element* 100
6.2.3 *Verification of the thermal element* 102
6.3 MECHANICAL MODEL 105
6.3.1 *Continuum mechanics* 105
6.3.2 *Finite element implementation of small strain elasto-plasticity* 111
6.3.3 *Verification of small strain elasto-plastic element* 116

7 FORMULATION OF MAGNETO-THERMAL-MECHANICAL PROBLEM 120

7.1 MAGNETO-THERMAL COUPLING 120
7.1.1 *Magneto-Thermal element* 120
7.1.2 *Verification of magneto-thermal element* 124
7.2 MAGNETO-THERMAL-MECHANICAL COUPLING 128
7.2.1 *Magneto-thermo-mechanical element* 128
7.2.2 *Verification of Magneto-Thermal-Mechanical element* 134
7.3 STAGED AND FULLY COUPLED SOLUTION STRATEGY 137

8 INDUCTION HEAT TREATMENT 145

9 CONCLUSIONS 153

A. APPENDIX 154

1 INTRODUCTION

1.1 Motivation and scope

During the last decade the range of problems, which are solved numerically using the finite element method has increased tremendously. Not only have new physical problems been introduced but also the complexity of the problems has risen. The computational models can nowadays include material and geometrical non-linearity, coupling of several fields, sensitivities etc...Derivation and testing of solutions for such type of problems is an error prone and time-consuming task.
In this work a solution environment based on a co-operative approach is presented which aims to provide flexible software tools for development and testing of computational models. Such an integrated environment, where the model code is generated, tested and applied can provide significant improvements of the development process.
The structure of the system is presented in Figure 1. It consist of the three major building blocks as follows:

- *AceGen*[1] - symbolic code generator
- *Computational Templates*[2] – interface modules for specific FE environment
- *Finite Element Driver*[2] – FE environment

The symbolic code generator AceGen utilizes several advanced techniques such as the symbolic and algebraic capabilities of *Mathematica*, automatic differentiation technique, automatic code generation, simultaneous optimization of expressions and theorem proving by the stochastic evaluation of expressions. Combinations of these techniques avoid the usual problem of uncontrollable growth of expressions known from other symbolic code generators. The *Computational Templates* module is an extension of the *AceGen* code generator. *Computational Templates* enables the generation of multi-environment finite element codes from the same abstract symbolic description. The third building block of the system is a finite element environment *Finite Element Driver*. It is not intended to be a replacement for

efficient general finite element environments, but to serve as a tool for the design of new numerical models and methods.
In the presented work the co-operative approach will be applied to the solution of coupled non-linear problems. The specific solution procedure will be addressed to an inductive heat treatment application.

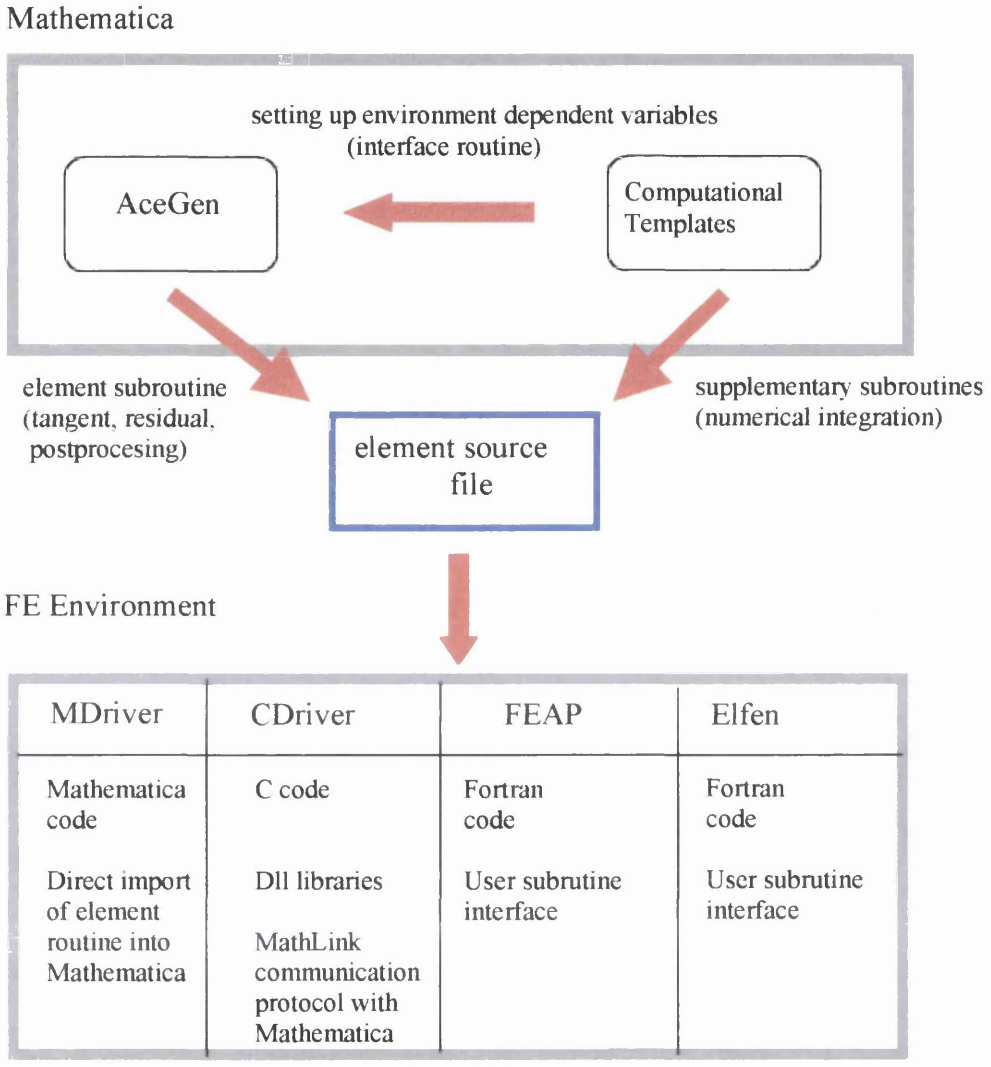


Figure 1 Schematic of the solution environment

1.2 Presentation order

In the second chapter the state of the art in scientific computing techniques and development trends of solution environments will be addressed while in the third chapter details of the co-operative solution approach will be provided.
In order to solve complex coupled problems the general abstract mathematical formulation of implicit solution methods for non-linear systems is presented in the fourth chapter. The high abstract level of the presented mathematical formulation is necessary in order to make symbolic formulation of continuum problems as general as possible. Numerical solutions were obtained using the finite element method, which will be presented in the fifth chapter.
The solution procedure will be applied to the problem of inductive heat treatment where the magnetic, thermal and displacement fields are coupled. Mathematical formulation, numerical model and its verification for each of individual fields will be presented in the sixth chapter. The development of the coupled element is discussed in the seventh while its application to the inductive heat treatment example is presented in chapter eight. Finally conclusions and recommendations for future work are given.
In the Appendix symbolic inputs for the inductive heat treatment example are provided.

References:

[1] J. Korelc, *Automatic generation of finite-element code by simultaneous optimalization of expressions*, Theoretical Computer Science, 187, p.p. 231-248, 1997.

[2] J. Korelc, *Hybrid system for multi-language and multi-environment generation of numerical codes*, Proceedings of the ISSAC'2001 Symposium on Symbolic and Algebraic Computation, New York, ACM:Press, 209-216,2001

2 FINITE ELEMENT SOLUTION ENVIRONMENTS

In the last decade computational methods have been applied to a wide range of application fields in science and engineering. The level of problem complexity has raised over the years and nowadays solutions of nonlinear, transient, coupled and path dependent problems are necessary to satisfy emerging application needs.
In order to meet new requirements the development of the finite element software should be focused on reduction of development and testing time utilizing new techniques. The modular structure of the finite element codes, which tends to be more open and flexible, also allows easier implementation of new developments.
The new approaches can be divided according to their role in the environment into two levels:

- element routine level
- FE environment level

The trend in the development of element routines goes towards automatic generation of code. The integration of several techniques which will be presented in subsection 2.1 is required to produce efficient element level routines. The advance in FE environment design goes toward increasing modularity and openness of the systems. Several available approaches to achieve these objectives will be presented in subsection 2.2.

2.1 State of the art in generation of FE codes

Reduction of the development time for the derivation of formulae and generation of characteristic quantities (**K**,**f**) can be achieved using the following state of the art techniques in scientific computing:

- Symbolic and algebraic approach
- Automatic differentiation
- Theorem proving
- Automatic code generation

2.1.1 SAC – symbolic and algebraic systems

Symbolic and algebraic systems are nowadays well-established tools in research and development. The most popular systems such as Mathematica[2] and Maple[3] contain a wide range of tools for general manipulation and calculation of formulas. Systems provide also a programming language, which allows further extension of the basic functionality. Special solutions, written in these languages are also available as separate add-on packages. In the case of finite element analysis several packages and courses are available primarily to the educational community [4][5].
Several papers have been published where SAC systems were successfully applied to the field of computational mechanics[17]-[20].
SAC systems are very powerful tools especially in the early stage of model development, where the mathematical formulation is tested and the range of validity is sought. Direct use of such systems for further development of complex FE models is not possible due to uncontrollable growth of expressions during the symbolic derivation of formulas. Also the numerical performance of SAC systems is lower than the performance of the available problem oriented solvers.

2.1.2 Automatic differentiation tools

Differentiation is a crucial arithmetic operation in the development of finite elements and hence the utilization of automatic differentiation tools can significantly reduce the development time. There are generally two methods available to obtain the computer code for evaluation of derivatives of function with respect to the chosen parameters: symbolic differentiation and automatic differentiation.
Symbolic differentiation is implemented inside SAC systems where the explicit expressions for derivatives are developed from the basic function. On the basis of the developed expressions the computer code is generated. The drawback of the symbolic differentiation method is that explicit expressions for derivatives can be derived only for a limited range of basic functions. If the basic function is of any significant complexity then the expressions for gradients tends to take several pages and therefore the generated code becomes inefficient.
The basic idea behind automatic differentiation[25][26] is to evaluate derivatives of the function from its computer code with respect to an arbitrary parameter. Therefore the input to the automatic differentiation algorithm is the computer code for evaluation of the function while the output is a code for evaluation of its derivatives with respect to selected parameters.
Extensive growth of expressions, which becomes a problem when using symbolic derivation, can be avoided using automatic differentiation techniques. Unfortunately the application to complex numerical models, as they appear in the case of finite

element analysis, is quite difficult. This is due to the fact that several implicit relations exist between the basic quantities and the iterative procedures that have to be performed on the level of individual finite elements as well as on global level. The amount of additional data can be as high as the number of numerical operations performed.
Many successful applications of this technique were reported using the ADIFOR[6] system, which uses FORTRAN code as an input. The ADIC[6] package was released providing automatic differentiation of C code.

2.1.3 Theorem proving systems

Theorem proving techniques can be applied to prove assumed relations inside a given set of formulas. This technique is essential for simplification of large expressions derived with SAC systems. Usually pattern-matching algorithms are used while in the case of mechanical problems theorem proving by examples[27] seems to be a better choice. At the present stage, TP systems are mostly used for proving geometric statements, which can be expressed as a set of algebraic formulae.

2.1.4 Automatic code generators

Automatic code generation is a technique, which is nowadays provided inside commercial symbolic environments. The equations are formed and manipulated inside SAC and at the end the code can be generated. Due to the uncontrollable growth of expressions arising from the problem complexity, only the code for relatively simple problems can be derived inside SAC.
Specialized stand-alone code generators were also designed targeting certain application fields or methods. Regarding generation of finite element code several attempts have been published[21]-[24] but none of them is able to treat expression swell within the automatic procedure.

2.2 State of the art in solution environments

Developers in the area of scientific computing are nowadays combining several computing techniques and software tools in order to build and test their models. Not only has the complexity of the problem increased but also the range of implementation problems, which they have to deal with. Therefore the progress in

scientific computing tends to provide software modularity in order to allow easier implementation of new developments. The modular structure can be provided using approaches such as object-oriented solutions, high-level problem solving environments (PSE) or hybrid systems. Especially in the case of coupled problems flexibility of the code architecture is a key issue due to complex communication, which can be required between several parts of the code.
Introduction of an object-oriented approach allows decomposition of the structural code into several objects communicating between each other. Such a concept can be far more open for introduction of new developments. The object-oriented approach reduces the amount of low level programming while usage of PSE introduces high-level programming languages that allows development of applications from prearranged modules. Usage of PSE for solutions of multi-field problems has increased due to the fact that solvers for various single field problems are available inside a single PSE as well as the data manipulation routines. There are several add-on packages already available, which are targeting multi-field problems. The hybrid environments consisting of several different applications with the steering application on top are also becoming important in the area of large scale and multi-field problems. From the current situation one can conclude that in the future open systems will play a major role in the scientific computing field.

2.2.1 Object oriented solutions

The object orientation strategy has become the major programming paradigm in software development strongly influencing also finite element development. The key features of object-oriented programming are [7]:

- Robustness and modularity
- Inheritance and polymorphism
- Non-anticipation and state encapsulation

The advantage introduced by object-orientation, storing the data into objects, offers a high level of modularity. The development of certain parts of the code without affecting other parts is provided by object independence and hence an expansion of the code functionality generally does not introduce new complexity to the code structure since the objects are modified not the system.
Another benefit of the approach is related to introduction of new developers, which can take over the maintenance, and development of the code without prior extensive experience with the system. At this point also the testing of object oriented code should be addressed which is limited to testing of individual object which is far less demanding then testing of the entire code.

There has been extensive discussion about the performance issues of object-oriented solutions but it has been shown[15] that the efficiency of C++ and FORTRAN numerical algorithms are comparable when certain implementation issues are taken into account. In the case of C++, CPU intensive numeric operations must take place in functions that are easily optimized by the compiler. Usage of advanced C++ syntax features, such as operator overloading, may drastically reduce the performance. To achieve better performance dynamic memory allocation and de-allocation should also be avoided.
The essential issue in development of object-oriented code is definition and organization of the object hierarchy. Several applications of object-oriented approaches have been presented in the literature [7]-[13].

2.2.2 Problem solving environments

Problem solving environments (PSE) are offering high level programming languages oriented to certain type of problems. The high-level program can be interpreted by a system or even the computer code for the particular application can be generated. Several PSE for solving partial differential equations by the finite element method are available such as Diffpack[13],SCIRun[30] and FlexPDE[28]. Usually the user templates are provided for a certain class of problems.
One of the most advanced PSE is object oriented Diffpack, which is organized as a collection of C++ libraries with classes, functions and utility programs. Diffpack is organized as a kernel product with a set of toolboxes providing specialized functionality. A wide variety of toolboxes is available including adaptive remeshing, parallelization and multigrid toolbox.
Following the approach that low-level, computationally intensive operations are performed in a FORTRAN or C style programming, while the object-oriented principles are mainly used for higher-level administrative tasks, Diffpack kernel provides high efficiency. While the kernel uses advantages of structural programming, the object oriented library (toolbox) offers more flexibility for functional extensions. Diffpack has a graphical user interface where the user can design a prototype application. Based on the prototype, an optimized version of the application can be generated suitable for running in a production context.
Numerical tools such, as MATLAB can be also very efficient on FE analysis offering a numerical matrix language. MATLAB package FEMLAB[14] is a general solution environment for partial differential equations. FEMLAB also features solution templates for a variety of multi-physic cases. At present additional FEMLAB packages for Chemical Engineering, Electromagnetics and Structural Mechanics are available.

Numerical libraries with compiled functions are still a common way of solving finite element problems and can be also grouped into PSE since in many cases they provide the foundation for high-level implementations. Adaptations of the code in order to communicate with the library functions are often time consuming and such solutions are not very flexible, but linear algebra libraries are still quite commonly used in FE environments due to their high numerical performances. The NAG[16] library is one of the most prominent.

2.2.3 Hybrid environments

The idea behind hybrid environments is to integrate different software tools into a single environment especially in the field of large scale and interdisciplinary computations. A single tool can be a particular problem solver, visualization application, mesh generator or even database engine.
A main role in the hybrid environment is the steering application. The steering application provides interfaces to the tools and it is responsible for program flow. In case of large-scale computations the steering application is also responsible for load balancing. The level of communication depends on the level of tool integration and its complexity. Examples of such tools are Alice[29] and SCIRun[30] which have been used for solution of several multi field problems where multiple models were used on different grids and discretizations.

References

[1] Ahmed K. Noor, *Computational structures technology: leap frogging into the twenty-first century*, Comupters & Structures, Vol 73, p.p. 1-31, 1999.

[2] *Mathematica*, Wolfram Research, Inc., http://www.wolfram.com/

[3] *Maple*, Waterloo Maple, Inc., http://www.maplesoft.com/

[4] University of Colorado, Center for Aerospace Structures, *Finite Element Programming with Mathematica*, MIFEM package : http://caswww.colorado.edu/courses.d/MIFEM.d/Home.html

[5] *ModelMaker package*, http://library.wolfram.com/conferences/devconf97/FiniteElement/

[6] Aragone National Laboratory, *Computational Differentiation Project*, http://www-unix.mcs.anl.gov/autodiff/

[7] S. Commend, T. Zimmermann: *Object oriented nonlinear finite element programming: a primer*, Advances in Engineering Software, Vol. 32, p.p. 611-628, 2001.

[8] Hededal O.,*Object oriented structuring of finite element*, Ph.D. thesis, Aalborg University, Denmark,1994.

[9] Rehak DR, Baugh JW Jr.,*Alternative programming techniques for finite element program development*, Procedings of the IABSE, Collegium on Expert Systems in Civil Engineering, Bergamo, Italy, 1989.

[10] Forde BWR, Foschi RO, Stiemer SF, *Object oriented finite element analysis*, Computers & Structures, Vol 34(3), p.p. 355-374,1990

[11] VectorSpace Programming, *VectorSpace C++ Library (vs.lib)*, http://vector-space.com/

[12] Aubry D.,Tie B., *Object Oriented programming in advanced structural computations*, ECCOMAS 2000, 2000.

[13] Numerical Objects AS, *Diffpack*, http://www.nobjects.com/Diffpack/

[14] Cosmol Group, *FEMLAB*, http://www.femlab.com/

[15] Arge E., Brauset A.M.,Clavin P.B., Kanney J.F., Langtangen H.P., Miller C.T., *On the Numerical Efficency of C++ in Scientific Computing*, Numerical Methods and Software Tools in Industrial Mathematics, M. Daehlen and A. Tveito (eds.),pp. 93-119,1997

[16] Numerical Algorithms Group, *NAG numerical libraries*, http://www.nag.co.uk/

[17] Iokamidis N.I., Anastasselos G.T., *Solution of plane elasticity problems with Mathematica*, Comuters and Structures, Vol. 55, No. 2, pp. 229-236,1995

[18] A. Eriksson, C. Pacoste, *Symbolic software tools in the development of finite elements*, Computers and Structures, 72,pp. 579-593,1999

[19] C.K. Yew, J.T. Boyle and D. MacKenzie, *Closed form integration of element stiffness matrices using a computer algebra system*, Computers and Structures, Vol. 56 No. 4,pp. 529-539,1995

[20] G. Amberg, R. Tonhardt, C. Winkler, *Finite element simulations using symbolic computing*, Mathematics and Computers in Simulation, 49, pp. 257 – 274, 1999

[21] Wang P.S., *Finger: a symbolic system for automatic generation of numerical programs in finite element analysis*, Journal of Symbolic Computations, 1986, 2,pp. 305-16

[22] Tan, H.Q.; Chang, T.Y.P.; Zheng, D.,*On symbolic manipulation and code generation of a hybrid 3-dimensional solid element*, Engineering with Computers, 7, 1, 47-59,1991

[23] Eyheramendy D., Zimmermann T., *Object oriented finite element programming: an interactive environment for symbolic derivations, application to an initial boundary value problem*, Advances in Engineering Software. Vol. 27, pp. 3-10, 1996

[24] Eyheramendy D., Zimmermann T.,*Object oriented finite elements II. A symbolic environment for automatic programming*, Comput. Methods Appl. Mech. Engrg., 132, pp. 277-304, 1996

[25] Griewank, A. (1989) *On Automatic Differentiation, Mathematical Programming: Recent Developments and Applications*, Kluwer Academic Publisher, Amsterdam, 83-108

[26] Bartholomew-Biggs, M; Brown, S.; Christianson, B.; and Dixon, L. *Automatic differentiation of algorithms*, Journal of Computational and Applied Mathematics, 124,1-2, 171-190,2000

[27] Gonnet, G., *New results for random determination of equivalence of expression*, Proc. of 1986 ACM Symp. on Symbolic and Algebraic Comp, (Char B.W., editor), Waterloo, 127-131,1986

[28] PDE Solutions Inc., *FlexPDE*, http://www.pdesolutions.com/

[29] Argonne National Laboratory, *Alice*, http://www-unix.mcs.anl.gov/alice/

[30] University of Utah, *SCIRun*, http://www.sci.utah.edu/research/pse_fields.html

3 CO-OPERATIVE APPROACH

In the presented work the co-operative approach[2]-[4] to the solution of continuum problems is used. The idea behind the co-operative approach is to combine several techniques inside one single environment instead of combining several different systems. This approach proves to overcome several drawbacks of the approaches presented in the previous section.

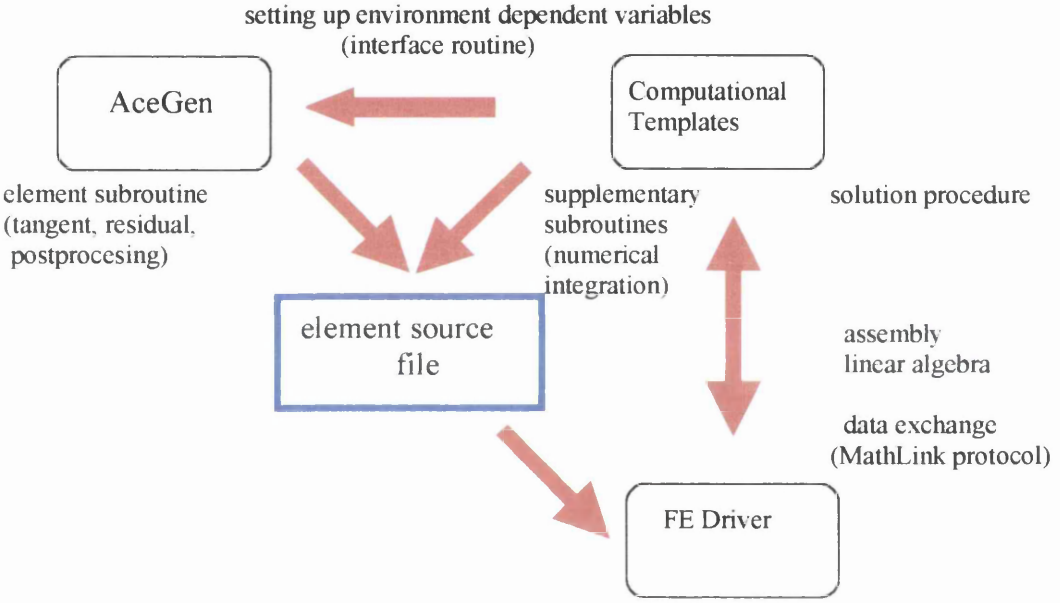


Figure 2 Schematic of the co-operative approach system

Choice of the algebraic computation system Mathematica to be the "master" system is natural due to the fact that the functionality provided inside such systems allows implementation of all other techniques. The system is presented in Figure 2 and it consists of three related packages:

- *AceGen*[6]
- *Computational templates*[7]
- *Finite Element Driver*

AceGen[6] is used for automatic derivation of formulae and code generation. The following techniques have been implemented into a single system: simultaneous

expression optimization, simultaneous generation of code structure and automatic differentiation. *Computational templates*[7] package, which is a collection of, prearranged modules for the automatic creation of interfaces between the generated finite element code and finite element environment. *Finite Element Driver* represents a model FE solution environment, which allows direct testing and application of the generated code. The *Computational templates* use *Finite Element Driver* assembly and linear algebra functionality in its solution procedures.

3.1 AceGen symbolic code generator

The symbolic code generator *AceGen* is based on the Simultaneous Stochastic Simplification[2] approach combining automatic differentiation and automatic theorem proving by examples with a general computer algebra system. Implementing the simultaneous simplification of expressions, based on stochastic evaluation of the formulae, extensive expressions growth and redundant calculations are avoided.
The general characteristics of the *AceGen* code generator are:

- simultaneous optimization of expressions immediately after they have been derived,
- automatic differentiation technique,
- automatic selection of the appropriate intermediate variables,
- the whole program structure can be generated,
- appropriate for large problems where also intermediate expressions can be subjected to uncontrolled swell,
- optimization procedures based on stochastic evaluation of expressions,
- generation of characteristic formulae,
- automatic interface to numerical environments using the *Compuational Templates* package,
- multi-language code generation (FORTRAN, C/C++, *Mathematica* language),
- advanced user interface,
- advanced methods for exploring and debugging of generated formulae,
- special procedures are needed for non-local operations.

The last characteristic mentioned is a consequence of restriction, which is imposed to non-local algebraic computations, such as integration, where the whole (global) expression has to be formed before applying the operation.

3.1.1 Simultaneous optimization

The classical way of optimizing expressions in computer algebra systems is searching for common sub-expressions at the end of the derivation, before the generation of the numerical code. In the numerical code common sub-expressions appear as auxiliary variables. An alternative approach is implemented in AceGen

where formulae are optimized, simplified and replaced by the auxiliary variables simultaneously with the derivation of the problem. The optimized version is then used in further operations. If the optimization is performed simultaneously, the explicit form of the expression is obviously lost, since some parts are replaced by intermediate variables.
In real problems it is almost impossible to recognize the identity of two expressions (for example the symmetry of the tangent stiffness matrix in nonlinear mechanical problems) automatically only by the pattern matching mechanisms. The only possible way at this stage of computer technology seems to be an algorithm that finds equivalence of expressions numerically. However, numerical identity is not a mathematically rigorous proof for the identity of two expressions. Thus the correctness of the simplification can be determined only with a certain degree of probability. With regard to our experience this can be neglected when dealing with more or less 'smooth' functions. In other cases, expressions have to be evaluated with a characteristic set of examples.

3.1.2 *AceGen* characteristic steps

Let us consider a simple example to illustrate the standard AceGen procedure for the generation of a typical numerical sub-program that returns the gradient of a given function f with respect to the set of parameters u_i. Let the unknown function u be approximated by a linear combination of unknown parameters u_1, u_2, u_3 and shape functions N_1, N_2, N_3 as follows

$$u = \sum_{i=1}^{3} u_i N_i$$

where $N_1 = x/L, N_2 = 1 - x/L$ and $N_3 = x/L(1 - x/L)$. The function f is defined as follows $f = u^2$.
The *AceGen* input can be divided into six characteristic steps as follows:

Step 1: Initialization

- The AceGen package is loaded and initialized. The C++ is chosen as the code programing language.

```
Get["AceGen`AceGen`", "user name"]
SMSInitialize["test", "Language" -> "C++", "Mode" -> "Optimal"];
```

Step 2: Definition of the input and output parameters

- This starts a new subroutine with the name "Test" and four real type parameters. The input parameters of the subroutine are vector u, and real numbers x, and L. Parameter g containing derivatives is an output parameter of the subroutine. The input and output parameters of the subroutine are characterized by the double $ sign at the end of the name.

```
SMSModule["Test", Real[u$$[3], x$$, L$$, g$$[3]]];
```

```
Module : Test
```

Step 3: Definition of Numeric-Symbolic Interface Variables

- Here the input parameters of the subroutine are assigned to the usual Mathematica variables. The standard Mathematica assignment operator = has been replaced by the special AceGen operator ⊨. Operator ⊨ performs stochastic simultaneous optimization of expressions

```
x ⊨ SMSReal[x$$]
L ⊨ SMSReal[L$$]
ui ⊨ Array[SMSReal[u$$[#1]] &, 3]
```

```
x
L
{ui1, ui2, ui3}
```

Step 4: Description of the Problem

- Here is the body of the subroutine.First the shape functions are defined.

```
Ni ⊨ {x/L, 1 - x/L, x/L*(1 - x/L)}
```

```
{Ni1, Ni2, Ni3}
```

- Unknown u is aproximated by $u = \sum_{i=1}^{3} N_i u_i$.

```
u ⊨ Ni . ui
```

```
u
```

- Function f is defined.

```
f ⊨ u^2
```

```
f
```

- This is where derivation of f with respect to parameters u_i is performed using automatic differen tiantion procedure implemented in the SMSD function.

```
g ⊨ SMSD[f, ui]
```

```
{g1, g2, g3}
```

Step 5: Definition of Symbolic - Numeric Interface Variables

- This assigns the results to the output parameters of the subroutine.

```
SMSExport[g, g$$];
```

Step 6: Code generation

- During the session AceGen generates pseudo-code which is stored into the AceGen database. At the end of the session AceGen translates the code from pseudo-code to the required script or compiled program language and prints out the code to the output file.

```
SMSWrite[];
```

```
Function : Test 5 formulae, 78 sub-expressions
[0] File created : test.c Size : 809
```

The generated source code file *test.c* follows.

```
/****************** S U B R O U T I N E ********************/
void Test(double v[501],double u[3],double *x,double *L,double
g[3])
{
v[6]=*x/*L;
v[7]=1e0-v[6];
v[8]=v[6]*v[7];
v[12]=2e0*(u[0]*v[6]+u[1]*v[7]+u[2]*v[8]);
g[0]=v[12]*v[6];
g[1]=v[12]*v[7];
g[2]=v[12]*v[8];
};
```

3.1.3 Assignment operators

A typical AceGen function takes the expression provided by the user, either interactively or in file, and returns an optimized version of the expression. The optimized version of the expression can result in a newly created auxiliary symbol v_i, or in an original expression in parts replaced by previously created auxiliary symbols. In the first case AceGen stores the new expression in an internal database. The database contains a global vector of all expressions, information about dependencies of the symbols, labels and names of the symbols, partial derivatives, etc. The database is a global object, which maintains information during the Mathematica session.

The AceGen system can generate three types of auxiliary variables: real type, integer type, and logical type auxiliary variables. Auxiliary variables have a standardized form *$V[i, j]*, where *i* is an index of an auxiliary variable and *j* is an instance of the *i*-th auxiliary variable. The new instance of the auxiliary variable is generated whenever a specific variable appears on the left hand side of equation. Although auxiliary variables are named consecutively, as they are entered by the user, they are not always stored in the database in the same order. Indeed, when two expressions contain a common sub-expression, AceGen immediately replaces the sub-expression with a new auxiliary variable, which is stored in the database in front of the considered expressions. The internal representation of the expressions in the database can be continuously changed and optimized.

The Mathematica functionality is extended with four additional assignment operators:

v = exp A new auxiliary variable is created if AceGen finds out that the introduction of the new variable is necessary, otherwise v=exp. This is the basic form for defining new formulae.

v := exp A new auxiliary variable is created, regardless of the contents of *exp*. The prime functionality of this form is to force the creation of a new auxiliary variable.

v =: exp A new auxiliary variable is created, regardless of the contents of *exp*. The prime functionality of this form is to create a variable, which will appear more than once on the left - hand side of equation.

v =! exp A new value *exp* is assigned to the previously created auxiliary variable *v*. At the input *v* has to be the auxiliary variable created as the result of the **v =: exp** command. At the output there is the same variable *v*, but with the new characteristic real values (new instance of *v*).

The last two assignment operators deal with auxiliary variables, which appears more that once on the left hand side of the expression. They are named "multi-valued variables" and are usually used within flow control constructs such as *If* and *Do* (see example in section 3.1.5).

3.1.4 Symbolic Input/Output Interface

A general way of how to pass data from the main program into the automatically generated routine and how to get the results back to the main program is trough external variables. External variables are used to establish the interface between the numerical environment and the automatically generated code.
External variables appear in a list of input/output parameters in the declaration of the subroutine, as a part of an expression, and when the values are assigned to the output parameters of the subroutine.
The input/output parameters to the routine are defined as follows:

$$SMSModule["name", Type1[p_{11},p_{12}, \ldots], Type2[p_{21},p_{22}, \ldots]]$$

Where "*name*" is a routine name with an argument block consisting of parameters specified by its type (*Type1, Type2*) and name in the parameter list. Valid types are:

Format	Description
$Real[p_{11},p_{12}, \ldots]$	List of real type parameters
$Integer[p_{11},p_{12}, \ldots]$	List of integer type parameters
$Logical[p_{11},p_{12}, \ldots]$	List of logical type parameters
$"typename"[p_{11},p_{12}, \ldots]$	List of the user defined type "typename" parameters

The form of the external variables is prescribed and is characterized by the $ signs at the end of its name. The standard *AceGen* form is automatically transformed into the

chosen language when the code is generated. The standard formats for external variables when they appear as part of subroutine declaration and their transformation into FORTRAN and C language declarations are as follows:

Variable type	*AceGen* definition	C definition	Fortran definition
Real variables	*x$$* *x$$$*	`double *x` `double x`	`real* 8 x` `real* 8 x`

Real arrays	*x$$[10]*
	x$$["10"]
	x$$[i$$, ""]*
	x$$[3, 5]
		`double x[10]`
		`double *x`
		`double **x`
		`double x[3][5]`
			`real* 8 x(10)`
			`real* 8 x(10)`
			`real* 8 x(i,*)`
			`real* 8 x(3,5)`

Integer variables	*i$* *i$$*	`int *i` `int i`	`integer* 8 i` `integer* 8 i`

Integer arrays	*i$$[10]*
	i$$[i$$, ""]*
	i$$[3,5,7]
		`int i[10]`
		`int **i`
		`int i[3][5][7]`
			`integer x(10)`
			`integer x(i,*)`
			`integer x(3,5,7)`

Logical	*l$* *l$$*	`int *l` `int l`	`logical l` `logical l`

For example:

SMSModule["sub1",Real[x$$,y$$[5]],Integer[i$$],Real[z$$],"mytype"[m$$]];

defines the routine named *sub1* with the following argument block:

C code :

```
void sub1(double v[501],double *x,double y[5],int *i,double
*z,mytype *m)
```

Fortran code:

```
SUBROUTINE sub1(v,x,y,i,z,m)
INTEGER i
DOUBLE PRECISION v(501),x,y(5),z
TYPE (mytype)::m
```

The standard format for external variables when they appear as part of the expression and their transformation into FORTRAN and C language formats is then:

Variable type	AceGen form	C form	Fortran form
Real variables	SMSReal[x$$] SMSReal[x$$$]	`*x` `x`	`x` `x`

Real arrays	SMSReal[x$$[10]]
	SMSReal[x$$["10"]]
	SMSReal[x$$[i$$, "->name",5]]
	SMSReal[x$$[i$$, ".name",5]]
		`x[10]`
		`x[10]`
		`x[i-1]->name[5]`
		`x[I-1].name[5]`
			`x(10)`
			`x(10)`
			`ilegal`
			`illegal`

Integer variables	SMSInteger[i$$] SMSInteger[i$$$]	`*i` `i`	`i` `i`

Integer arrays	SMSInteger[i$$[10]]
	SMSInteger[i$$["10"]]
	SMSInteger[i$$[j$$,"->name",5]]
	SMSInteger[i$$[j$$, ".name",5]]
		`i[10]`
		`i[10]`
		`i[j-1]->name[5]`
		`i[j-1].name[5]`
			`i(10)`
			`i(10)`
			`ilegal`
			`ilegal`

Logical	SMSLogical[l$$] SMSLogical[l$$$]	`*l` `l`	`l` `l`

In order to export any regular expressions the external variables have to be used. At the end of the session, the external variables are translated into the FORTRAN or C format using the *SMSExport* command as follows:

SMSExport[value, external variable]

where the value and external variable have to be of the same type.
For example:

SMSModule["test",Real[x$$]]

produces the following C code function declaration:

```
void test(double v[501],double *x)
```

In order to evaluate the square of x with the function *test* the following steps are required:

- local variable x is created and assigned the value of external variable x
  $x \triangleright$ *SMSReal[x$$]*
- the square of the local variable x is exported to external variable x
  SMSExport[x^2, x$$]

Therefore after evaluation of function *test* ,variable x contains its square.

3.1.5 Flow control operators

AceGen can automatically generate conditionals (*SMSIf*, *SMSElse* and *SMSEndIf*) and loops (*SMSDo* and *SMSEndDo*). The program structure specified by the conditionals and loops is created simultaneously during the AceGen session and it will appear as a part of automatically generated code in a specified language. In addition, whole parts of code can be inserted using the *SMSVerbatim* command.
In case of conditionals the same auxiliary variable can appear in separate branches of the *If* statement. In such a case the multi-valued variables are used to prevent incorrect simplifications. Typical usage is presented by the following example.

Consider the generation of a procedure, which evaluates the following function

$$f(x)=\begin{cases} x\le 0 & x^2 \\ x>0 & \sin x \end{cases}$$

In this example **f** will be represented as a multi-valued variable.

```
SMSInitialize["test", "Language" -> "C++"];
SMSModule["test", Real[x$$, f$$]];
x = SMSReal[x$$];
SMSIf[x <= 0];
 f = x^2;
SMSElse[];
 f = Sin[x];
SMSEndIf[f];
SMSExport[f, f$$];
SMSWrite[];
```

The resulting C++ code is presented below.

```
/******************* S U B R O U T I N E *********************/
void test(double v[501],double *x,double *f)
{int b2;
v[1]=*x;
if(v[1]<=0e0){
 v[3]=(v[1]*v[1]);
} else {
 v[3]=sin(v[1]);
};
*f=v[3];
};
```

3.1.6 Automatic differentiation

The automatic differentiation procedure implemented in *AceGen* uses a vector of auxiliary variables, generated during the simultaneous simplification as a pseudo code for automatic differentiation. Explicit parts of the expression are differentiated using Mathematica's symbolic differentiation capabilities.

Higher order derivatives are difficult to implement by standard automatic differentiation tools and most of the automatic differentiation tools offer only the first derivatives. When AceGen derives derivatives, the results and all the auxiliary formulae are stored in a global vector of formulae where they act as any other formula entered by the user. Thus, there is no limitation in AceGen concerning the number of derivatives, which are to be derived.

The function *SMSD[exp, var]* performs automatic differentiation of one or several expressions with respect to an arbitrary variable or vector of variables by

simultaneously enhancing the already derived code. In the following cases additional steps should be taken to obtain proper derivatives:

- in the case of implicit dependency between variables v_1 and v_2 the derivative $\partial v_1 / \partial v_2$ should be explicitly defined by the following command:
  SMTDefineDerivative[$v_1, v_2, \partial v_1 / \partial v_2$ *]*
- if the explicit dependencies of *exp* should be neglected during the differentiation then *SMSFreeze[exp, "Dependency"]* command must be applied.
- in the case of other exceptions where the evaluation of the derivative code would lead to numerical errors *SMSFreeze* can also be applied to specific exceptions.
  SMSFreeze[exp, "Dependency",{{ $v_1, \partial \exp / \partial v_1$*},{*$v_2, \partial \exp / \partial v_2$*},...}]*

3.1.7 AceGen user interface

Ace Gen provides an advanced user interface, which allows the user to explore the structure of the derived formulas.
AceGen at startup displays the palette (see Figure 3) consisting of buttons, which allows the user to change between different output representations of the expressions and to explore polymorphism of the generated formulae.

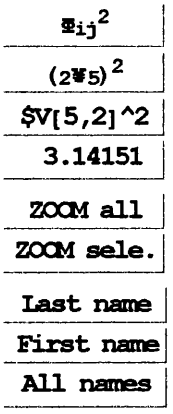
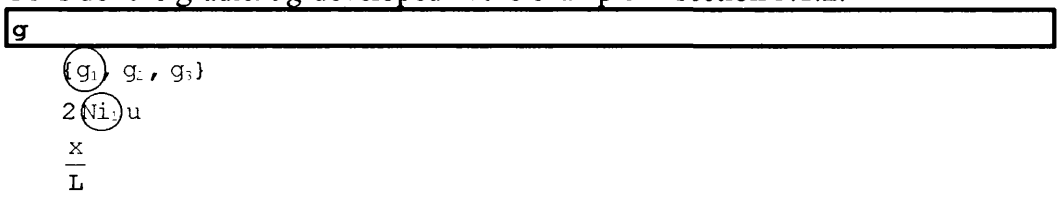


Figure 3 *AceGen* button palette

Auxiliary variables are represented in Mathematica's notebook as active areas (buttons) of the output form of the expressions in blue color. When we point with the mouse to one of the active areas, a new cell in the notebook is generated and the definition of the pointed variable will be displayed. Auxiliary variables are again represented as active areas and can be further explored. Definitions of the external

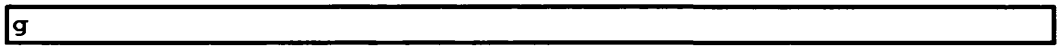
variables are displayed in red colour. Only the main features of the user interface related to the presented work will be addressed.

Consider the gradient *g* developed in the example in section 3.1.2.



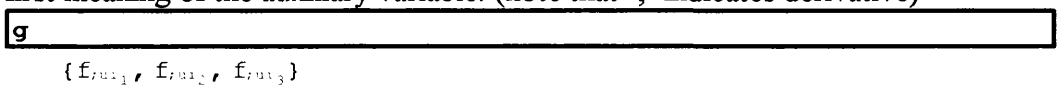
Selecting Σ_{ij}^2 and entering *g* we get its representation as $\{g_1,g_2,g_3\}$ which is an automatically generated name. Pointing with the mouse (red circle) to g_1 displays its definition. Pressing the mouse button while pointing to N_1 displays its definition.

By selecting $V[5,2]^2$ the variable *g* is represented by its full form displaying the actual form of variables in the database

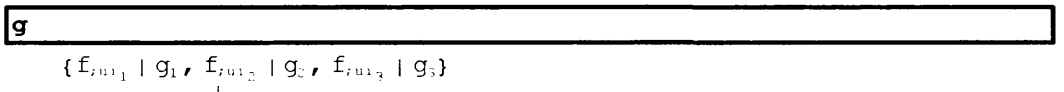


In an automatically generated source code the *i*-th term of the global vector of auxiliary variables (*v(i)*) directly corresponds to the *$V[i,j]* auxiliary variable

In order to explore the polymorphism the First name button allows finding of the first meaning of the auxiliary variable. (note that ';' indicates derivative)



All meanings (names) of the auxiliary variables can also be explored using the All names button.



The Last name button is the default setting and displays the last name and hence the output is of the form $\{g_1,g_2,g_3\}$ as discussed in the first case.

3.2 Computational templates

The code generated by *AceGen* is a generic numerical code, which is suitable for implementation in any numerical environment. In order to include generated code directly into certain numerical environments, modification of the code should be performed. Usually the adaptation of data structures is required or new interface routines have to be written in order to establish communication between different parts of the code. The other possibility is to provide additional functionality to the code generator to produce the code, which can be directly plugged into a particular environment. The *Computational Templates* provides such a functionality to the *AceGen* code generator allowing use of the same symbolic input for generation of elements, which can be used inside different finite element environments.

Interfacing the automatically generated code with a finite element environment can be quite a difficult process. Taking the routine for evaluating element stiffness as an example it is clear that the arguments to the routine contain the same information while the syntax and the programming language can be different. The *Computational Templates* package overcomes the problem by adding environment dependent code to the generated subroutine, providing proper data transfer between subroutine and environment. For each environment the separate specific file has to be written containing additional interface code.

Additionally to the FE interface module, *Computational Templates* also contains a set of solution procedures, which uses functionality of the model FE environment called *Finite Element Driver*. The *Finite Element Driver* provides assembly and linear algebra functionality to the *Computational Templates*.

The advantage of this model environment is that it exists in two equivalent codes. The first version is written in Mathematica's symbolic language. Thus when a particular problem is analyzed, the advantages of Mathematica such as high precision arithmetic, interval arithmetic, or even symbolic evaluation of FE quantities can be used. The second version is written in C language providing better numerical performances so that large-scale problems can be solved at the same time. At the exploratory stage the efficient element code is generated from the same symbolic description and incorporated into the professional finite element environment. The *Computational Templates* system currently supports generation of code for the following FE environments: *MDriver* is a model FE environment written in a Mathematica's symbolic language, *CDriver* is a model FE environment written in a C language, FEAP is a research environment written in FORTRAN, ELFEN is a commercial environment written in FORTRAN.

Elements created with *Computational Templates* are self-sufficient and hence no external subroutines are called from the element routine. All the data is exchanged through subroutine argument blocks.

3.2.1 Characteristic example

The syntax of *Computational Templates* is the same as the syntax of *AceGen* with an additional set of commands added. The set of *Computational Templates* commands starts with *SMT* while the *AceGen* commands starts with *SMS*. The *Computational Templates* characteristic example will be presented first.
As an example the three-dimensional element for steady state heat conduction will be generated. The steady state heat conduction is defined by

$$\frac{\partial}{\partial x}\left(k\frac{\partial T}{\partial x}\right)+\frac{\partial}{\partial y}\left(k\frac{\partial T}{\partial y}\right)+\frac{\partial}{\partial z}\left(k\frac{\partial T}{\partial z}\right)+Q=0 \tag{3.1}$$

on domain Ω where T is the temperature and k is the thermal conductivity. The boundary conditions are $T=T_o$ on Γ_T and $k\frac{\partial T}{\partial n}=q_o$ on Γ_q . The term $\frac{\partial T}{\partial n}$ is a directional derivative defined as $\frac{\partial T}{\partial n}=\frac{\partial T}{\partial x}n_x+\frac{\partial T}{\partial y}n_y+\frac{\partial T}{\partial z}n_z$ and q_o is the prescribed value of surface flux. The thermal conductivity k is assumed to be the quadratic function of temperature: $k(T)=k_0+k_1T+k_2T^2$.
The weak formulation of equation (3.1) obtained by the Galerkin approach is as follows

$$\int_\Omega[(\nabla^T\delta T)k\nabla T-\delta TQ]d\Omega-\int_{\Gamma_q}\delta T\,q_o\,d\Gamma=0 \tag{3.2}$$

The problem considered is non-linear and it has an un-symmetric tangent matrix.
The symbolic input can be divided into the following steps:

Step 1: Initialization

- The AceGen and Computational Templates packages are loaded and initialized. C++ is chosen as the code programing language. CDriver is selected as the target FE environment.

```
Get["AceGen`AceGen`", "user name"]
Get["ComputationalTemplates`SMT`"];
SMSInitialize["heat_conduction", "Language" -> "C++",
  "Mode" -> "Prototype"];
SMTInitialize["heat_conduction", "CDriver", "SMTTopology" → "H1",
              "SMTDOFGlobal" → 1, "SMTSymmetricTangent" → 0];
```

Step 2: Element subroutine for the evaluation of tangent matrix and residual

- Start the definition of the user subroutine for the calculation of the tangent matrix and residual vector and set up of input/output parameters for the CDriver environment.

```
SMTUserSubroutine["Tangent and residual"];
```

Step 3: Interface to the input data of the element subroutine

- Here the coordinates of the element nodes and current values of the nodal temperatures are taken from the supplied arguments of the subroutine.

```
Xi ⊨ Array[ SMSReal[nd$$[#, "X", 1]] &, 8];
Yi ⊨ Array[ SMSReal[nd$$[#, "X", 2]] &, 8];
Zi ⊨ Array[ SMSReal[nd$$[#, "X", 3]] &, 8];
Ti ⊨ Array[SMSReal[nd$$[#, "at", 1]] &, 8];
```

- The conductivity parameters k0,k1,k2 and the internal heat source Q are assumed to be common for all elements in a particular domain (material or group data).

```
SMTGroupDataNames = {"Conductivity parameter k0",
   "Conductivity parameter k1",
   "Conductivity parameter k2", "Heat source"};
{k0, k1, k2, Q} ⊨
  SMSReal[{es$$["Data", 1], es$$["Data", 2], es$$["Data", 3],
    es$$["Data", 4]}];
```

- Element is numerically integrated by one of the built-in standard numerical integration rules. This starts the loop over the integration points, where ξ, η, ζ are coordinates of the current integration point and wGauss is the integration point weight.

```
SMSDo[IpIndex, 1, SMSInteger[es$$["id", "NoIntPoints"]]];
{ξ, η, ζ, wGauss} ⊨ Array[SMSReal[es$$["IntPoints", #1, IpIndex]] &, 4];
```

Step 4: Definition of the trial functions

- This defines the trilinear shape functions N_i, $i=1,2,...,8$ and interpolation of the physical coordinates within the element. J_m is the Jacobian matrix of the isoparametric mapping from actual coordinate system X, Y, Z to reference coordinates ξ, η, ζ. The implicit dependencies between the actual and the reference coordinates are given by $\frac{\partial \xi_j}{\partial X_i} = J_m^{-1} \frac{\partial X_i}{\partial \xi_j}$, where J_m is the Jacobian matrix of the nonlinear coordinate mapping.

```
Ni ⊨ MapThread[ 1 / 8 (1 + ξ #1) (1 + η #2) (1 + ζ #3) &,
    Transpose[{{-1, -1, -1}, {1, -1, -1}, {1, 1, -1}, {-1, 1, -1},
      {-1, -1, 1}, {1, -1, 1}, {1, 1, 1}, {-1, 1, 1}}]];
X ⊨ SMSFreeze[Ni.Xi];
Y ⊨ SMSFreeze[Ni.Yi];
Z ⊨ SMSFreeze[Ni.Zi];
Jm ⊨ SMSD[{X, Y, Z}, {ξ, η, ζ}];
SMSDefineDerivative[{ξ, η, ζ}, {X, Y, Z}, SMSInverse[Jm]];
```

- The trial function for the temperature distribution within the element is given as linear combination of the shape functions and the nodal temperatures $T = N_i.T_i$. The terms T_i are unknown parameters of the variational problem.

```
T ⊨ Ni.Ti;
```

Step 5: Definition of the governing equations

- Here is the definition of the weak form of the steady state heat conduction equations.

```
k ⊨ k0 + k1 T + k2 T^2;
δT ⊨ SMSD[T, Ti];
Φi ⊨ Det[Jm] wGauss (k SMSD[δT, {X, Y, Z}].SMSD[T, {X, Y, Z}] - δT Q);
```

- Element contribution to the global residual vector is exported to the *p$$* output parameters of the "*Tangent and residual*" subroutine.

```
SMSExport[SMTResidualSign Φi, p$$, "AddIn" → True];
```

Step 5: Definition of the tangent matrix

- This evaluates the explicit form of the tangent matrix and exports result into the s$$ output parameter of the user subroutine. Another possibility would be to generate a characteristic formula for the arbitrary element of the residual and the tangent matrix. This would substantially reduce the code size.

```
Kij = SMSD[ei, Ti];
SMSExport[Kij, s$$, "AddIn" → True];
```

- This is the end of the integration loop.

```
SMSEndDo[];
```

Step 6: Code generation

- At the end of the session *AceGen* translates the code from pseudo-code to the required script or compiled program language and prepares the interface for the generated code according to the content of the *SMTSplice file*.The result is *heat_conduction.c* file with the element source code written in C.

```
SMSWrite["Splice" → SMTSplice];
```

The same input can be used for the generation of the code for other environments. The only required modifications are in the initialization step, where the language and environment variable have to be set in *SMTInitialize* according to Table 1.

FE Environment	Environment variable	Language
MDriver	"MDriver"	"Language" -> "Mathematica"
CDriver	"CDriver"	"Language" -> "C++"
FEAP	"FEAP"	"Language" -> "Fortran"
ELFEN	"ELFEN"	"Language" -> "Fortran"

Table 1 Computational Templates – environment specifications

In order to illustrate the difference between generated code for different environments the inputs for *MDriver*, *CDriver* and FEAP are presented. The splice file additions are clearly presented while the bodies of the subroutines are empty since they contain the same code.

MDriver code *: heat_conduction.m*

```
SMT`SetElSpec["heat_conduction",idata$$_,ic_,gd_]:=Block[{q1,q2,q3},
q3=SMTMultiIntegration[ic];
q1={"heat_conduction",
{"SKR" -> SMT`SKR, _ -> Null}
,{"SpecIndex",3,8,0,8,4,30,
 ic,"NoTimeStorage",0,q3[[1]],0,0,0,q3[[3]],q3[[4]],q3[[5]]},"H1",
 {"Conductivity parameter k0", "Conductivity parameter k1",
"Conductivity parameter k2", "Heat source"},
 {},
 {},
 {1, 2, 3, 4, 0, 2, 6, 5,
 1, 0, 1, 4, 8, 5, 0, 4,
 3, 7, 8, 0, 3, 2, 6, 7,
```

```
 0, 7, 8, 5, 6, 0},
 {1, 1, 1, 1, 1, 1, 1, 1},{},{},gd,q3[[2]]//Transpose};
q1[[3,9]]=0;q1];

(********************** M O D U L E *************************)
SetAttributes[SMT`SKR,HoldAll];
SMT`SKR["heat_conduction",es$$_,ed$$_,nd$$_,rdata$$_
     ,idata$$_,p$$_,s$$_]:=Module[{},
  ];
```

CDriver code : *heat_conduction.c*

```
#include "sms.h"
void SKR(double v[505],ElementSpec *es,ElementData *ed,NodeData
**nd,
   double *rdata,int *idata,double *p,double **s);
FILE *SMTFile;

__declspec(dllexport) void SMTSetElSpec(ElementSpec *es,int
*idata,int *ic,double *gd)
{ static int pn[30]={1, 2, 3, 4, 0, 2, 6, 5,
             1, 0, 1, 4, 8, 5, 0, 4,
             3, 7, 8, 0, 3, 2, 6, 7,
             0, 7, 8, 5, 6, 0};
  static int dof[8]={1, 1, 1, 1, 1, 1, 1, 1};
  static char *gdcs[]={"Conductivity parameter k0",
  "Conductivity parameter k1","Conductivity parameter k2","Heat
source"};
  static char *gpcs[]={""};
  static char *npcs[]={""};
  es->Code="heat_conduction";
  es->id.NoDimensions=3;es->id.NoDOFGlobal=8;
  es->id.NoDOFCondense=0;es->id.NoNodes=8;
  es->id.NoGroupData=4;es->id.NoSegmentPoints=30;
  es->id.IntCode=*ic;es->id.NoElementData=0;
  es->Segments=pn;es->DOFGlobal=dof;
  es->Data=gd;es->id.NoGPostData=0;
  es->id.NoNPostData=0;es->id.SymmetricTangent=0;
  es->IntPoints=SMTMultiIntPoints
     (ic,idata,&es->id.NoIntPoints,&es->id.NoIntPointsA,
     &es->id.NoIntPointsB,&es->id.NoIntPointsC);
  es->id.NoTimeStorage=0;
  es->Topology="H1";es->GroupDataNames=gdcs;
  es->GPostNames=gpcs;es->NPostNames=npcs;
  es->user.SKR=SKR;
};
/******************* S U B R O U T I N E ********************/
void SKR(double v[505],ElementSpec *es,ElementData *ed,NodeData **nd
     ,double *rdata,int *idata,double *p,double **s)
```

FEAP code : *heat_conduction.f*

```
      subroutine elmt10(d,ul,xl,ix,tl,s,p,ndfe,ndme,nste,isw)
      implicit none
      include 'sms.h'
      integer ix(nen),ndme,ndfe,nste,isw
      double precision  xl(ndfe,nen),d(*),ul(ndfe,nen,*)
      double precision  s(nste,nste),p(nste),tl(nen),sxd(24)

      double precision ul0(ndfe,nen),sg(8),sg0(8)
      character*50 SELEM,datades(5),postdes(0)
      logical DEBUG
      parameter (DEBUG=.false.,
     # SELEM="heat_conduction")
      integer i,j,jj,ll,ii,k,kk,i1,i2,i3,hlen,icode
      double precision w,v(505),gpost(64,0),npost(8,0)
      integer ipordl(30)
      data (ipordl(i),i=1,30)/1,2,3,4,1,2,6,5,1,2,1,4,8,5,1,
     & 4,3,7,8,4,3,2,6,7,3,7,8,5,6,7/
1235  format(i3,">",1g17.9)
1236  format("   >",5g17.9)
1237  format(i3,"> ",8g11.5)
1238  format(i3,"> ",8g11.5)

      if(isw.ne.1) then
         icode=int(d(5))
         if(idata(ID_LastIntCode).ne.icode) then
          call SMSIntPoints(icode,ngpo,gp)
          idata(ID_LastIntCode)=icode
         endif
      endif
      do i=1,ndfe
        do j=1,nen
           ul0(i,j)=ul(i,j,1)-ul(i,j,2)
        enddo
      enddo
      idata(ID_Iteration)=niter+1
      if(nstep.eq.1) then
         idata(ID_TotalIteration)=niter+1
      else
         idata(ID_TotalIteration)=1000
      endif
      rdata(RD_SubIterationTolerance)=1d-9
      go to(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,
     #      18,19,20,21), isw
c
c.... input record 1 of material properties
1     call dinput(d(1),5)
      icode=int(d(5))

      call SMSIntPoints(icode,ngpo,gp)
      write(*,*)  SELEM
      write(iow,*)SELEM
c.....Description of the input data
```

```
      datades(1)="Conductivity parameter k0"
      datades(2)="Conductivity parameter k1"
      datades(3)="Conductivity parameter k2"
      datades(4)="Heat source"
      datades(5)="Integration code"

      write(iow,"(10x,f15.5,A3,A50)")
     #     (d(i)," = ",datades(i),i=1,5)
c.... number of history variables

      idata(ID_NoSensParameters)=0
      nsenpa=idata(ID_NoSensParameters)
      mct=0
c.....number of data for TECPLOT
      ntecdata=-5
c.... define node numbering for plot mesh routine, see pltord
      inord(10) = 30
      do ii = 1,30
       ipord(ii,10) = ipordl(ii)
      end do
c.... number of projected postprocess quantities
      istv=-11
c.... description of the postprocessing data

      idata(ID_OutputFile)=iow
      return
2     write(*,*)"User switch 2 not implemented"
      return
3     continue
c.... tangent and residuum

      call SKR10(v,d,ul,ul0,xl,s,p,hr(nh2),hr(nh1))

      return
4     continue
      goto 8
5     write(*,*)"User switch 5 not implemented"
      return

6     goto 3
7     write(*,*)"User switch 7 not implemented"
      return
c.... postprocessing
8     continue
c.....Description of the post-processing data

      return
9     continue
10    continue
11    continue
12    continue
13    continue
c..... initialize history
14    return
```

```
15    continue
16    continue
17    continue
18    continue
19    continue
      write(*,*)"User switch ",isw," not implemented"
      return
c.....sensitivity analysis - external
20    continue

      return
c..... internal sensitivity
21    continue

      return
      End

******************** S U B R O U T I N E **********************
      SUBROUTINE SKR10(v,d,ul,ul0,xl,s,p,ht,hp)
      IMPLICIT NONE
      include 'sms.h'
      INTEGER i37
      DOUBLE PRECISION
v(505),d(4),ul(1,8),ul0(1,8),xl(3,8),s(8,8),p(8
     &),ht(*),hp(*)
      END
```

3.2.2 *Computational Templates* interface commands

As mentioned in the introduction *Computational Templates* consist of two distinctive parts. The first part deals with interfacing the generated code to a specific environment while the second part represents the solution procedures for the FE environment called *Finite Element Driver*.

First the interface commands, used in the characteristic example, will be briefly described.

As in the case for *AceGen* case the initialization step is performed first by the *SMTInitialize* command with the following syntax:

SMTInitialize[element_name, environment, options]

where *element_name* is an arbitrary name, the environment variable is set according to Table 1 while options are listed in Table 2.

Option	Comment
SMTTopology	element topology code
SMTNoDimensions	number of spatial dimensions
SMTNoNodes	number of element nodes
SMTDOFGlobal	number of global DOF per node
SMTSymmetricTangent	flag indicating tangent matrix symmetry

SMTGroupDataNames	description of the input data values that are common for all elements with the same element specification
SMTGPostNames	description of the post-processing quantities defined per integration point
SMNPostNames	description of the post-processing quantities defined per nodal point
SMTNoDOFCondense	number of DOF that have to be statically condensed before the element quantities are assembled
SMTNoTimeStorage	total number of history dependent real type values per element that have to be stored in the memory for transient type of problems
SMTNoElementData	total number of arbitrary real values per element
SMTNoDOFGlobal	total number of global DOF
SMTMaxNoDOFNode	number of DOF per node used for dimensioning local arrays
SMTNoAllDOF	number of DOF used for dimensioning local arrays
SMTResidualSign	sign of the residual depending on the equation forming (in case $K a + \Psi = 0 \rightarrow 1$, $K a = \Psi \rightarrow -1$)
SMTSegments	for all segments on the surface of the element the sequence of the element node indices that define the edge of the segment
SMTNodeOrder	ordering of nodes when compared to the standard ordering

Table 2 *Computational Templates* – element options.

Although there is quite a long list of options many of them are set automatically to default values.
After the package is initialized the actual user subroutine has to be specified. Definition of the user subroutine is performed using the *SMTUserSubroutine* command with the following syntax:

SMTUserSubroutine[code,name,arg]

Code	**Default Name**	**Arg**
Tangent and residual	*SKR*	*p$$[NoDOFGlobal]* *s$$[NoDOFGlobal,NoDOFGlobal]*
Postprocessing	*SPP*	*gpost$$[NoIntPoints,NoGPostData]* *npost$$[NoNodes,NoNPostData]*
Sensitivity pseudo-load	*SSE*	*p$$[NoDOFGlobal]*
Dependent sensitivity	*SHI*	
Residual	*SRE*	*p$$[NoDOFGlobal]*
User n	*Usern*	

Table 3 *Computational Templates* – element subroutine options.

The details about specific functions will be provided in the following sections.
In addition the command *SMTPrint[c, f, expr1,expr2,...]* is provided which can be used for generation of the source code for printing out certain expressions specified by *expr1, expr2* etc... . The flag *c* specifies whether the message should be printed to *stdout* while the *f* flag specifies output to a default output file.
As presented in the example, the definitions of element formulation and derivation of the element characteristic quantities (tangent matrix and residual) are performed using standard *AceGen* commands while the code generation step differs since the splice file has to be specified to *SMSWrite* command (*SMSWrite["Splice"→SMTSplice]*). The *SMTSplice* file contains the previously discussed environment dependent code additions. It is clear that for customization of the generated code for new environments the proper splice file has to be provided.

3.3 Finite Element Driver

Finite Element Driver represents the model FE environment. The term 'finite element driver' stems from the correlation of the present approach with some standard software products where basic functionality is provided by an independent kernel. Different drivers are then used to adapt the output of the kernel to various platforms and input-output devices. In our case, the symbolic code generator provides the basic problem dependent functionality (finite elements) while the remaining finite element code at the global level is reduced to the "finite element driver" that consists of iteration loop, solver, pre-post processing and platform dependent codes.
In the case of the present system the solution procedures are part of the *Computational Templates* while the *Finite Element Driver* represents the independent computational kernel. Two versions of the *Finite Element Driver* environment are implemented to date as presented on Figure 4. The first one is called *MDriver* and is written in Mathematica's script language while the second version known as *CDriver* is written in ANSI C and it communicates with Mathematica via the MathLink[5] protocol.

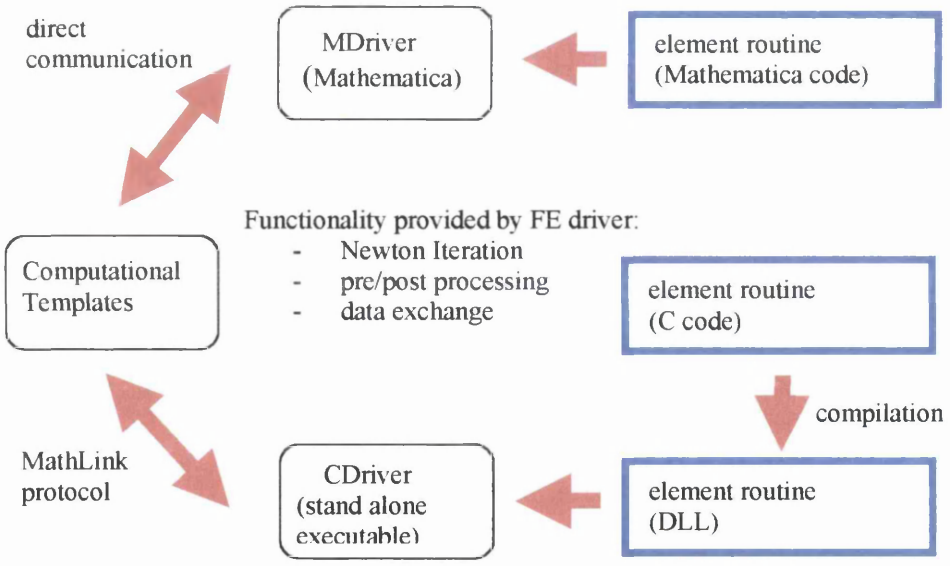


Figure 4 Scheme of the *Finite Element Driver* concept

Both drivers share the same data structures and support the same set of commands and hence the same input can be used for both versions. The version of the driver, which is going to be used, is selected at the beginning of the analysis using a single

command. The most important difference between the two versions is in their numerical performances. When a particular problem is analyzed, the advantages of Mathematica such as high precision arithmetic, interval arithmetic, or even symbolic evaluation of FE quantities can be used within *MDriver*. While the *MDriver* can be successfully applied to small problems during a development stage its performance drops significantly with growing scale of the problem. Thus for large-scale problems the *CDriver* can be used, providing better numerical efficiency.
The specific use of the element routines within *FE Driver* should be emphasized. The element routines are specified in the analysis input, as they are not part of the driver code. In the *MDriver* case the element are loaded directly from their source files since no compilation is required. In the *CDriver* case the element source code is compiled into a dynamic linked library (DLL). The dynamic library is then loaded into memory as a separate module on input request. In this case the linking of the element routine with the driver source code is not required since the element dynamic library acts as an independent module.
In this work the *CDriver* implementation will be presented in detail.

3.3.1 Characteristic example of *Finite Element Driver* usage

The functionality of the *Finite Element Driver* can be best explained by an example. The finite element analysis of the problem, using the heat conduction element derived in the previous section, will be presented.
Consider the example (see Figure 5) of heat conduction in a single element with prescribed temperatures at nodes 1,2,3,4 (red points) and prescribed heat flux at nodal point 7 (yellow point).

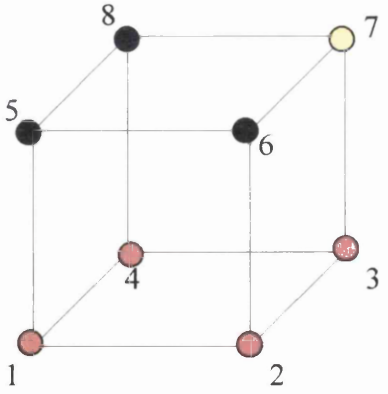


Figure 5 FE Driver – geometry of characteristic example

Characteristic steps of finite element analysis using *Finite Element Driver* are as follows:

Step 1: Initialization

- Environment start-up

```
SMTSetDriver["CDriver"];
```

Step 2: Geometry and mesh definiton

- Here the symbols are defined that describe the node coordinates, the element connections and the boundary conditions.

```
SMTNodes = {{1, -0.5, 0, -0.5}, {2, 1, 0, 0}, {3, 1, 1.5, 0}, {4, 0, 1.5, 0},
   {5, 0, 0, 1.1}, {6, 1, 0, 1}, {7, 1.35, 1, 1}, {8, 0, 1, 1}};
SMTNaturalBoundary = {{7, 5.5}};
SMTEssentialBoundary = {{1, 0.1}, {2, 0.2}, {3, 0.3}, {4, 1.1}};
SMTElements = {{1, 1, {1, 2, 3, 4, 5, 6, 7, 8}}};
```

- This is where the element is chosen by defining the element specification. Specification is defined by giving: the name of the element, the source code file with the element subroutines, the element specification data k0,k1,k2 and the integration code.

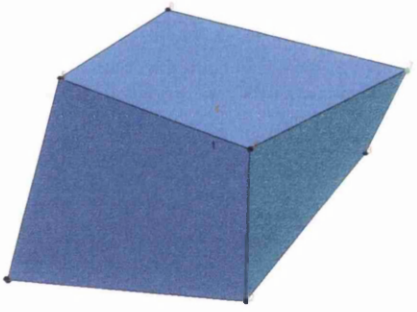
```
SMTElementSpec =
  {{1, "heat_conduction", "heat_conduction", {10., .5, .1, 1.}, 7}};
```

- This checks the input data, imports the element source code and starts the analysis.

```
SMTStructure[];
```

- The mesh can be visualised including boundary conditions

```
SMTShowMesh["BoundaryConditions" → True, "Marks" → True];
```



Step 3: Analysis

- Here the real time and the value of the boundary condition multiplier are prescribed. The problem is steady-state so that the real time in this case has no meaning.

```
SMTNextStep[0, 1];
```

- Here the problem is solved by the standard quadratically convergent Newton-Raphson iterative method. The observed quadratic convergence is also a proof that the problem was correctly linearized. This test can be used as one of the code verification tests.

```
While[SMTNewtonIteration[] > 10^-14, SMTStatusReport[SMTNodeData[5, "at"]]];
SMTStatusReport[SMTNodeData[5, "at"]]
```

```
Time=0. Multiplier=1. Iteration=1
 ||Δa||=2.20066 ||Ψ||=7.33364 Status=0 Tag={0.713743}
Time=0. Multiplier=1. Iteration=2 ||Δa||=
0.0741088 ||Ψ||=0.285841 Status=0 Tag={0.705225}
Time=0. Multiplier=1. Iteration=3 ||Δa||=
0.000102125 ||Ψ||=0.000406596 Status=0 Tag={0.70522}
Time=0. Multiplier=1. Iteration=4 ||Δa||=
1.76657×10^-10 ||Ψ||=7.1198×10^-10 Status=0 Tag={0.70522}
Time=0. Multiplier=1. Iteration=5 ||Δa||=
4.85721×10^-16 ||Ψ||=1.88677×10^-15 Status=0 Tag={0.70522}
```

- During the analysis we have at all times full access to the entire environment, nodal and element data. They can be accessed and changed with the data manipulation commands which gives to drivers flexibility that is not shared by other FE environments.

Here an additional step is made however,the boundary condition in node 3 is changed.

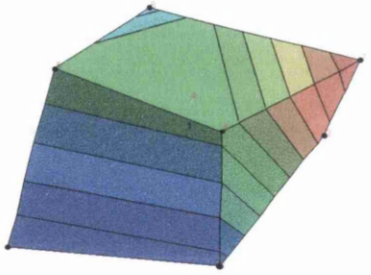
```
SMTNextStep[1, 1];
SMTNodeData[3, "Boundary", {1.5}];
While[SMTNewtonIteration[] > 10^-14,];
SMTStatusReport[SMTNodeData[5, "at"]];
```

```
Time=1. Multiplier=2. Iteration=6 ||Δa||=
4.84383×10^-16 ||Ψ||=2.39407×10^-15 Status=0 Tag={1.98632}
```

Step 4: Post-processing

- Here the SMTShowMesh function displays a three-dimensional contour plot of the current temperature distribution.

```
SMTShowMesh["BoundaryConditions" → True, "Marks" → True,
  "ElementValues" → SMTPost[1], "Contour" → True];
```



During the execution of the analysis, which is controlled from Mathematica, any *Finite Element Driver* data structure can be accessed and also modified using the data manipulation functions. From the user point of view, these functions are independent of the solution environment (CDriver either MDriver). The environment

is specified using the *SMTSetDriver* command (CDriver is the default environment) the remainder of the input is the same for all the analysis.

3.3.2 *Finite Element Driver* basic data structures

The concept of *Finite Element Driver* is based on the transparent basic data structures. The basic data structures are as follows:

- Environmental data
- Element specification data
- Element data
- Node data

The structures implemented in *CDriver* are identical to those in the *MDriver* environment. To get a better insight each structure and its manipulation function will be described in detail. In the description the *Finite Element Driver* generic data will be addressed in the form of the *AceGen* external variables (see section 3.1.4). The external variables are characterized by the $ signs at the end of its name.

3.3.2.1 Environmental data

The general information related to the status of the solution procedure is stored in the environmental data. The environmental data are stored in two vectors. The *IData* vector contains all the integer type values while *RData* stored all the real type values. The fields in both vectors can be accessed via the predefined code (constants).
The detailed structure of *IData* is presented in Table 4 while the *RData* structure is presented in Table 5.

idata$$["code"]	***Description***
idata$$["IDataLength"]	length of *IData* vector
idata$$["RDataLength"]	length of *RData* vector
idata$$["IDataLast"]	index of the last value reserved on *IData* vector
idata$$["RDataLast"]	index of the last value reserved on *RData* vector
idata$$["LastIntCode"]	last integration code for which numerical integration points and weights were calculated
idata$$["Iteration"]	index of the current iteration within the iterative loop
idata$$["TotalIteration"]	total number of iterations in session
idata$$["LinearEstimate"]	if 1 then in the first iteration of the Newton-Raphson iterative procedure the prescribed boundary conditions are not updated.

idata$$["ErrorStatus"]	contains the code for the type of error event if any
idata$$["MaterialState"]	number of "Non-physical material point state" error events detected from the last error check
idata$$["NoSensParameters"]	total number of sensitivity parameters
idata$$["ElementShape"]	number of "Non-physical element shape" error events detected from the last error check
idata$$["SensIndex"]	global index of the current sensitivity parameter
idata$$["OutputFile"]	output file number or output channel number
idata$$["MissingSubroutine"]	number of "Missing user defined subroutine" error events detected from the last error check
idata$$["SubDivergence"]	number of "Divergence of the local sub-iterative process" error events detected from the last error check
idata$$["ElementState"]	number of "Non-physical element state" error events detected from the last error check
idata$$["NoNodes"]	total number of nodes
idata$$["NoElements"]	total number of elements
idata$$["NoESpec"]	total number of element specifications
idata$$["Debug"]	if 1 debug messages are written in the output file
idata$$["NoDimensions"]	number of spatial dimensions of the problem
idata$$["SymmetricTangent"]	1 if global tangent matrix is symmetric else 0
idata$$["NoTmpStore"]	minimum number of real type variables per node which have to be stored temporarily
idata$$["NoEquations"]	total number of global equations
idata$$["DiagonalSign"]	number of "Solver: change of the sign of diagonal" error events detected from the last error check
idata$$["Task"]	code of the current task performed
idata$$["NoSubIterations"]	maximal number of local sub-iterative process iterations performed during the analysis
idata$$["CurrentElement"]	index of the current element in process
idata$$["MaxPhysicalState"]	used for the indication of the physical state of the element (e.g. 0-elastic, 1-plastic, etc...)
idata$$["ExtrapolationType"]	type of extrapolation of integration point values to nodes. If 1 then integration point value is multiplied by the shape function value. If 0 least square extrapolation is used.

Table 4 *Finite Element Driver –IData* structure

The data stored in *IData* can be accessed and modified using the function *SMTIData*. The syntax of the function is as follows: *SMTIData["code",value]* where the code specifies the *IData* entry, according to Table 3, while the *value* represents the new value of the variable. If the *SMTIData* function is used only with *"code"* argument then the current value of the variable is returned. This rule applies to all data manipulation functions within *Finite Element Driver*.

rdata$$["code"]	*Description*
rdata$$["Multiplier"]	current values of the natural and essential boundary conditions are obtained by multiplying initial values with the multiplier
rdata$$["ResidualError"]	Euclid's norm of the residual vector
rdata$$["IncrementError"]	Euclid's norm of the last increment of global DOF
rdata$$["MFlops"]	estimate of the number of floating point operations per second
rdata$$["SubMFlops"]	number of equivalent floating point operations for the last call of the user subroutine
rdata$$["Time"]	real time
rdata$$["TimeIncrement"]	value of the last real time increment
rdata$$["MultiplierIncrement"]	value of the last multiplier increment
rdata$$["SubIterationTolerance"]	tolerance for the local sub-iterative process
rdata$$["StepSizeControl"]	step size control factor

Table 5 *Finite Element Driver –RData* structure

As in the *IData* case the *SMTRData* function is provided for manipulation of the *RData* vector. The syntax and the rules of command are identical to *SMTIData* except the *"code"* field has to be in agreement with Table 5.
Example of *SMTRdata* and SMTIData usage:

SMTRData["ResidualError"]

returns the Euclidian Norm of the current residual vector.

SMTIData["OutputFile", "report.txt"]

sets the analysis output file to file *report.txt.*

3.3.2.2 Element Specification

ElementSpec is the most complex data structure implemented in *Finite Element Driver* and it is equivalent to the *ElementType* structure, which is often used in finite element programs. Structure of the *ElementSpec* is presented in Table 6.

es$$["code"]	***Description***
es$$["Code"]	element code according to the general classification (string)
es$$["User",i]	the i-th user defined element subroutines. This field represents the link to a specific routine. (environment specific)
es$$["id","SpecIndex"]	global index of the specification
es$$["id","NoDimensions"]	number of spatial dimensions
es$$["id","NoDOFGlobal"]	number of global DOF
es$$["id","NoDOFCondense"]	number of DOF that have to be statically condensed before the element quantities are assembled to global quantities
es$$["id","NoNodes"]	number of element nodes
es$$["id","NoGroupData"]	number of input data values that are common for all elements with the same element specification (material characteristics, etc...)
es$$["id","NoSegmentPoints"]	the length of the es$$["Segments"] field
es$$["id","IntCode"]	integration code
es$$["id","NoTimeStorage"]	number of transient variables
es$$["id","NoElementData"]	number of arbitrary real values per element
es$$["id","NoIntPoints"]	number of integration points
es$$["id","NoGPostData"]	number of integration point post-processing quantities
es$$["id","NoNPostData"]	number of nodal point post-processing quantities
es$$["id","SymmetricTangent"]	if 1 the element tangent matrix is symmetric else 0
es$$["id","NoIntPointsA"]	number of integration points for first integration code
es$$["id","NoIntPointsB"]	number of integration points for second integration code
es$$["id","NoIntPointsC"]	number of integration points for third integration code
es$$["Topology"]	element topology code
es$$["GroupDataNames",i]	description of the i-th input data value that is common for all elements with the same specification (list of strings)

es$$["GPostNames",i]	description of the i-th post- processing quantities evaluated at each integration point (list of strings)
es$$["NPostNames",i]	description of the i-th post- processing quantities evaluated at each nodal point (list of strings)
es$$["Segments",i]	sequence of element node indices that defines the segments on the surface or outline of the element
es$$["DOFGlobal",i]	number of DOF for the i-th node (each node can have different number of DOF)
es$$["SensType",i]	type of the i-th sensitivity parameter
es$$["SensTypeIndex",i]	index of the i-th parameter defined locally in a type group
es$$["Data",i]	data common for all the elements with a particular element specification
es$$["IntPoints", i]	coordinates and weights of the i-th numerical integration point

Table 6 *Finite Element Driver –ElementSpec* structure

The fields within *ElementSpec* structure can be accessed and modified using the *SMTElementSpec* function with the following syntax:

SMTElementSpec[i,code]	returns the value of the field *code* in the *i*-th specification
SMTElementSpec[i,code,j]	sets the value of the field *code* in the *i*-th specification to value *j*
SMTElementSpec[code]	returns the value of the field *code* for all specifications.

Examples of *SMTElementSpec* usage:

SMTElementSpec[1," Data"]

returns the vector of material and other group data for first specification in the list

SMTElementSpec[2,"Data",{10, 7.12 10^6, 0.3}]

sets the vector of material and other group data for second specification in the list and returns the list of current values.

3.3.2.3 Element data

The element data structure is presented in Table 7.

ed$$["code"]	*Description*
ed$$["id", "ElemIndex"]	global index of the element
ed$$["id", "SpecIndex"]	index of the element specification
ed$$["id", "Active"]	current element status (1 if active)
ed$$["Nodes", j]	number of the *j*-th element node
ed$$["Data", j]	arbitrary element specific data
ed$$["ht", j]	current state of the *j*-th transient element specific variable
ed$$["hp", j]	the state of the *j*-th transient variable at the end of the previous step

Table 7 *Finite Element Driver –ElementData* structure

The fields within the *ElementData* structure can be retrieved and modified using *SMTElementData* function with the same syntax rules as in the case of the *SMTElementSpec* command.
The *SpecIndex* code specifies the index of the element specification. Based on this index all the element group data and corresponding functions can be retrieved from the corresponding *ElementSpec* entry in the list of element specifications. The element connectivity table (list of element nodes) can be obtained from the *Nodes* field. *Data* field can be used for various data which are not history related such as information about condensation of local DOF, initial strains etc....
Example of *SMTElementData* usage:

SMTElementSpec[SMTElementData[55,"SpecIndex"],"Code"]

returns the name of the element specification for element number 55.

SMTElementSpec[102,"ht",{1.0,1.0,1.0}]

sets the values of fields in history vector to 1 for element number 102.

3.3.2.4 Node data

nd$$[i, "code"]	*Description*
nd$$[i, "id", "NodeIndex"]	global index of the *i*-th element
nd$$[i, "id", "NoDOF"]	number of nodal DOF
nd$$[i, "id", "Fictive"]	flag identifying the node role (1 if physical, 0 if auxillary)
nd$$[i, "DOF",j]	global number of *j*-th DOF
nd$$[i, "X",j]	*j*-th initial spatial coordinate

nd$$[i, "Boundary",j]	if nd$$[i,"DOF",j] = -1 then the value of the j-th essential boundary condition else value of j-th natural boundary condition
nd$$[i, "at",j]	current value of the *j*-th nodal DOF
nd$$[i, "ap",j]	value of the *j*-th nodal DOF at the end of previous step
nd$$[i, "da",j]	current value of the *j*-th nodal DOF increment
nd$$[i, "st",j,k]	current sensitivities of the *k*-th nodal DOF with respect to the *j*-th sensitivity parameter
nd$$[i, "sp",j,k]	sensitivities of the *k*-th nodal DOF with respect to the *j*-th sensitivity parameter in previous step
nd$$[i,"tmp", j]	temporary real type variables stored during the execution of a single analysis directive

Table 8 *Finite Element Driver –NodeData* structure

Each node can be identified through the *id* vector, which contains its number, number of degrees of freedom and the flag that indicates its mesh status. The *DOF* vector contains the list of nodal global degrees of freedom (DOF). Nodal coordinates are stored in the vector *X* whose length depends on the case dimension (2 in 2D or 3 in 3D). Temporary storage field *tmp* can be used for various purposes such as the storage of testing parameters, introduction of new parameters into the model etc... The *NodeData* structure by itself is not related to any other data structure and represents the basic data structure of the *Finite Element Driver*.
The *SMTNodeData* is the data manipulation function related to *NodeData* structure. The same syntax rules are applied as in other data manipulation functions.

3.3.2.5 Relations between the basic data structures

The data organization of the *Finite Element Driver* is very simple and transparent. The basic data structure relations are presented in Figure 6.

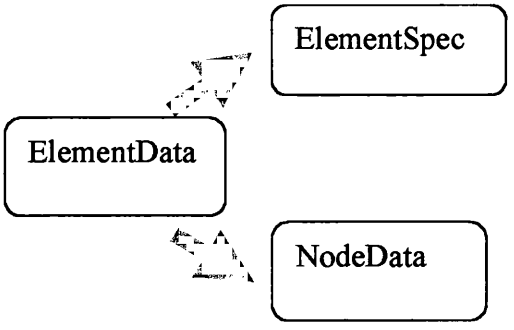


Figure 6 *FE Driver* – Relations between data structures

The basic data types are as independent as possible. The *ElementSpec* and *NodeData* structure are completely independent while the *ElementData* structure is related to both.
The *ElementData* structure is related to its corresponding *ElemeSpec* through the *ElementData ed$$["id","SpecIndex"]* field while the *ElementData ed$$["Nodes"]* field relates it to the *NodeData* structure. The *ed$$["Nodes"]* contains the list of global element node numbers corresponding to a particular element. The data relations are in both cases unidirectional.
The data model presented with minimal data structure interconnections is the key feature of the *Finite Element Driver* concept allowing high level of modularity.

3.3.3 Finite Element Driver problem input data

The data concerning the problem mesh is stored into five global vectors.
First the node vector *SMTNode* is formed in the following form:

$$SMTNodes = \{node_1, node_2, node_3, \ldots, node_{NoNodes}\}$$

Each *node* entry should be in the format $node_i = \{ inode, X, Y, Z\}$ where *inode* is the node number and X,Y and Z are its spatial coordinates. In the case of two-dimensional problems the coordinate Z is not required.
Elements are stored into the *SMTElements* vector:

$$SMTElements = \{elem_1, elem_2, elem_3, \ldots, elem_{NoElements}\}$$

with characteristic entry

$$elem_i = \{ ielem, iespec, \{ inode_1, inode_2, \ldots \}$$

where *ielem* is the element global number, *iespec* is the index of corresponding element specification and $\{ inode_1, inode_2, \ldots \}$ is the list of element nodes also known as the element connectivity table.
Element specifications vector *SMTElementSpec* is of the following form:

$$SMTElementSpec = \{spec_1, spec_2, spec_3, \ldots, spec_{NoESpec}\}$$

and

$$spec_i = \{ iespec, etype, scode, \{ d_1, d_2, \ldots, d_{NoGrupData} \}, intcode \}$$

where *iespec* is the global element specification index, *etype* element identifying code (string), *scode* is the element source file (string) ,$\{ d_1, d_2, \ldots, d_{NoGrupData} \}$ list of group data (material data etc..) and *intcode* is the integration code.

Boundary conditions are stored in two vectors *SMTEssentialBoundary* and *SMTNaturalBoundary*. Essential boundary entry should be of the following form:

$$pnode_i=\{ inode, v_1, v_2, v_3, \ldots, v_{NoDOF})$$

where *inode* denotes the node number while v_i is the essential boundary condition for the *i*-th degree of freedom. In the case that the value is not prescribed the *Null* symbol should be used. Instead of specifying the node numbers the *condition* form can be applied as follows: $pnode_i=\{ condition, v_1, v_2, v_3, \ldots, v_{NoDOF})$ where all the nodes for which the condition, in terms of spatial coordinates, is fulfilled the boundary condition is applied. The conditional form is applicable for the case of essential and natural boundary conditions.
The natural boundary entry is essentially similar to the condition entry:

$$nnode_i=\{ inode, f_1, f_2, f_3, \ldots, f_{NoDOF})$$

with f_i as the natural condition for the *i*-th degree of freedom. The *Null* symbol is again used for an unspecified value.

There are two possibilities how to enter the problem mesh into the *Finite Element Driver*. The first possibility is to use the *Computational Templates* structural mesh generator. The mesh generator is written in Mathematica and is available through *SMTStructuredMesh* command that constructs *SMTNodes* and *SMTElements* input data arrays for a structured mesh of elements. Basic syntax is as follows:

SMTStructuredMesh[array, division, topology]

where array represents the regular two or three dimensional array of the arbitrary number of points that outlines the boundary of the problem domain. The division parameter defines a number of elements in each direction while the element type is defined by the topology parameter. The *SMTStructuredMesh* is limited to creation of structured meshes, which are limited to relatively simple geometries.
In the case of complex geometries, where unstructured meshes are required other mesh generators have to be used. In this work the GiD preprocessor[8], which offers flexible customization options was used for creation of complex geometries and related unstructured meshes. The preprocessor was customized in order to produce meshes that can be directly imported into *Finite Element Driver*. Due to simplicity of the mesh input file almost any mesh generator can be used. The structure of the mesh input file will be presented briefly.

3.3.3.1 Finite element mesh input file

Finite Element Driver reads mesh input from the ASCII file, which can be directly created using *Computational Templates* commands or using the GiD preprocessor. Format of the ASCII input file for the *CDriver* is divided into six parts as follows:

1. File header
2. Finite element specification
3. Element connectivity
4. Node coordinates
5. Essential boundary condition
6. Natural boundary condition

3.3.3.1.1 File Header

First line of the mesh input file contains five integer values specifying the number of dimensions of the problem, number of mesh nodes, number of elements and number of different finite element specifications and number of sensitivity parameters. According to the header data mesh vectors are allocated.

Line 1: *NoDimensions NoNodes NoElements NoSpecificatons NoSensitivities*

Example :

2 4 1 1 4 2D problem, 4 nodes, 1 element, 1 element specification, 4 sensitivity parameters

3.3.3.1.2 Finite element specification

The specification block contains at least one entry describing a particular finite element specification. Each entry consists of two lines.
First line contains the finite element specification code (string):

Line 1: *Element Specification code*

In the second line the data related to the specification is specified. Number of element specification, integration code and number of material parameters are integers while material parameters are treated as real numbers.

Line 2: *ElementSpecNo IntCode NoGroupData GroupData*

Example:

PEDFQ1Q1EPSIsoHookeMissesC
1 2 4 1.1 3000 0.2 100

The element to be used is *PEDFQ1Q1EPSIsoHookeMissesC* with specification index equal to 1, integration code is 2, there are 4 group parameters of the following values 1.1 (thickness), 3000 (Elastic modulus), 0.2 (Poisson ratio) and 100 (uniaxial yield stress)

3.3.3.1.3 Element connectivity

The element connectivity table contains one entry per element. The element entry consists of the following integers:

Line 1: *ElementSpecNo Node_List*

The node list contains a list of element nodes. The number of entries in the *Node_List* list is checked according to the number of nodes set by the corresponding element specification. Length of the table (number of rows) must be equal to *NoElements* specified in the file header.

3.3.3.1.4 Nodes coordinates

Nodes coordinates are specified by one entry per node in the following format:

Line 1: *NodeNo NoCoordinates CoordinatesList*

The *NodeNo* and *NoCoordinates* are integers while the space coordinates of the node, which are specified in the *CoordinatesList*, are real numbers.

Number of entries (number of rows) must be equal to *NoNodes* specified in the file header.

3.3.3.1.5 Essential boundary condition

This part of the input file must start with the line containing one single integer specifying the number of essential B.C. entries, which is equal to the number of nodes with prescribed value of the global degree of freedom.

Line 1: *NoEssentialBC*

Each entry contains:

Line 2: *NodeNo EBCFlagList EBCValueList*

NodeNo is an integer specifying the node index. Restriction of a certain degree of freedom is prescribed in *EBCFlagList* (one integer per DOF) with a flag which can be equal to 1 (if the DOF is restricted) or to 0 (if the DOF is not restricted). Prescribed values are specified in *EBCValueList* as *NoDOF* real numbers (one real number per DOF).

Example:
In the case of an element with two global degrees of freedom (for example displacement u and v)
4 1 1 0. 0. Node number 4 has prescribed value of 0 to both global degrees of freedom

3.3.3.1.6 Natural boundary condition

Block of the input file containing the natural boundary condition data must also start with the line containing one single integer specifying the number of entries.

Line 1: *NoNaturalBC*

Each entry contains:

Line 2: *NodeNo NBCValueList*

The *NodeNo* is an integer specifying the node index while the *NBCValueList* contains the prescribed values of load.
Example:
In the case of a displacement based element the force can be prescribed to a certain node.

9 0. 150. The node 9 has applied force in the force 150 in the *y* direction.

3.3.4 *Finite Element Driver* analysis

In previous sections the basic data structures and manipulation functions were presented as well as the mesh input format. Once the input is specified the analysis can be performed.
First the analysis should be initialized by reading the input data. According to the input header the required memory can be allocated for global vectors and the data can be transferred from input into memory. The initialization step is performed using the *SMTStructure[ ]* command, which reads the input data arrays, creates the analysis input file and start the analysis. The command accepts input file as an argument in the case of external mesh generation as follows *SMTStructure[Input→"mesh.inp"]* where *mesh.inp* is the name of the file containing the mesh data. The *SMTStructure* also imports all the element source files (*MDriver*) or in the case of *CDriver* compiles the element source files and creates dynamic link library files (DLL file) with the element subroutines. It is worthwhile to note that the element code is automatically compiled only in the case when the DLL library does not exist or in the case when the newer version of the code is generated.
The other important task, which is performed by SMTStructure, is initialization of nodal degrees of freedom according to support data so that the global degree of freedom numbering can be established. Once the number of global unknowns is recognized the linear algebra structures are allocated as well.
Once the entire problem is transferred from input to the proper data structures the analysis can be started. The Newton-Raphson incremental method will be implemented as a solution procedure.
The first step of analysis is to increase the time and boundary conditions multiplier which is done by the *SMTNextStep[time_increment, multiplier_increment]* command. After applying the "load" the solution can be sought by an iteration process where in each step a global system of equation is solved until the convergence criterion is satisfied.
A Single Newton-Raphson iteration step is performed by the *SMTNewtonIteration[]* command which performs the following steps:

- values of the boundary conditions are multiplied with the environment constant "Multiplier" (*rdata$$["Multiplier]*)
- user subroutine "Tangent and residual" is called for each element
  - the element tangent matrix is added to the global matrix K,
  - element residual is added to the global vector Ψ
- the set of linear equations $\mathbf{K}\Delta\mathbf{a}=\Psi$ is solved for increment $\Delta\mathbf{a}$
- solution is incremented $\mathbf{a}_t=\mathbf{a}_t+\Delta\mathbf{a}$

SMTNewtonIteration[] returns the Euclidian norm $\|\Delta\mathbf{a}\|$ which can be used as a convergence criterion.

The Newton-Raphson iteration step undertakes a loop of the following form where *SMTNewtonIteration[]* is executed until the increment norm is lower than the convergence limit or the number of iteration reaches a certain limit:

```
While[SMTNewtonIteration[] > 10^-9 && SMTIData["Iteration"] < 10,
  SMTStatusReport[]];
```

If convergence is achieved then analysis can proceed to the next load step executing *SMTNextStep[]* which also makes the solution at the end of the previous time step to be equal to the current solution (**ap=at,hp=ht**).

In cases when the convergence is not obtained the state of the analysis should be returned to the last convergent solution using *SMTStepBack[]* which maps the converged solution to the current solutions (**at=ap,ht=hp**) in the opposite way to the *SMTNextStep[]* case. The time and multiplier should be reduced and a new iteration can be started.

The *SMTStatusReport[]* is used to print out the current status of the system.

3.3.5 Finite Element Driver post-processing

The visualization of the analysis results can be performed in several different ways depending on the type of visualization.

The most popular option is to present the results on the finite element mesh. In that case the *SMTShowMesh* is available offering a wide range of visualization options which are discussed in Table 9. *SMTShowMesh* uses the graphic capabilities of Mathematica.

Before discussing the visualization option the organization of post-processing data should be addressed. The function, which provides the post-processing data, is *SMTPost.* The syntax varies depending on the type of data. The following forms are used:

SMTPost[i_Integer]	returns a vector composed of the *i*-th nodal degree of freedom from all nodes
SMTPost[pcode_String]	returns a vector composed of the nodal values according to element post-processing code *pcode*

The second form of the function can only be used if the element user subroutine for data post-processing has been defined. The post-processing code *pcode* can access either the integration point or the nodal point post-processing quantity. The integration points are mapped to nodal using extrapolation rule set by the *idata$$["ExtrapolationType"].*

Option	*Description*
Mesh	display mesh as wire frame
Marks	display nodal and element numbers
BoundaryConditions	mark nodes with the prescribed essential boundary condition with color points and nodes with the non-zero natural boundary condition with arrows
Elements	fill in the element surfaces with the element surface colour or with the contour lines
ElementValues	the vector of nodal values p that defines the scalar field to be visualized
NodeValues	the vector of nodal values n that are used for visualization at each nodal point mark
Contour	display contour lines of the scalar field p defined by the "ElementValues" option
DeformedMesh	display deformed mesh by adding the displacements vector field **u** multiplied by the "Scale" option to the initial nodal coordinates
Scale	scaling factor for deformed mesh
Legend	include legend specifying the colours and the range of the "ElementValues" values
IncludeElements	list of the element specification indexes to be plotted
Label	label for the plot
TextStyle	specifies the default text style and font
NodeMarks	mark all the nodes with a circle

Table 9 *Finite Element Driver –SMTShowMesh* command options

The *SMTShowMesh* uses the following syntax:

SMTShowMesh["option"→ value]

where the value has to be of the proper type. In the case where no arguments are specified the command returns a mesh with the elements coloured according to their element specification.

The most important option of the *SMTShowMesh* command is *ElementValues* which supplies the element visualization field. *SMTShowMesh* uses the *SMTPost* function to obtain the required data as follows:

SMTShowMesh["ElementValues"→ SMTPost[pcode_String]]

where the post-processing quantity is specified using the SMTPost command directly.

Another very useful processing function is the *SMTPointValues[p, nv]* function which evaluates a scalar field defined by the nodal values *nv* at point *p*.

The distribution of a certain nodal quantity within a profile can be easily obtained as for example:

```
profile = {Table[x, {x, -b, b, b/50.}],
   SMTPointValues[Table[{a, x}, {x, -b, b, b/50.}], {SMTPost[2]}] //
    Flatten};
```

where the *profile* stores the distribution of the second degree of freedom on a line, from (a,-b) to (a,b), using 50 sampling points.
The *Finite Element Driver* post-processing functions in combination with Mathematica's table and list operations provide a very flexible environment for visualization and analysis of results.
The usage of an external post-processor is also possible since results can be exported to ASCII files. In the case of an imported mesh from the GiD preprocessor we might also use its post-processing capabilities. In that case we can use GiD specific output command *SMTGIDWriteResults["postcode"]*, which writes directly to the GiD post-processing file the result specified by the *postcode* string as in the case of *SMTPost*.

3.3.6 *CDriver* implementation details

The *CDriver* is implemented in ANSI C. All the data structures and functions presented in previous sections are implemented providing the full compatibility with Mathematica's *MDriver* environment and hence from the user point of view there is no real difference working with *MDriver* or *CDriver*.
In the following sections the important implementation details will be addressed which were necessary to achieve the compatibility objective. First the data communication protocol will be presented then an outline of the functions will be given and at the data structures will be described.

3.3.6.1 CDriver - MathLink interface

The Mathematica package offers a protocol called *MathLink* for communication of the external program with the Mathematica kernel. The link can be established in both directions offering Mathematica functionality to the external program as well as access from Mathematica to the external program.
In the *CDriver* case all the data can be accessed from Mathematica. Once the data is being transferred from *CDriver* to *Mathematica* it can be modified, using Mathematica's full functionality and transferred back to CDriver.
In order to establish the communication between the external application and *Mathematica* via *MathLink* protocol, the application has to be linked with the *MathLink* library. The *MathLink* library provides a set of C functions for exchange of data streams between the application and *Mathematica*. These functions are used in *CDriver* to exchange the data with *Mathematica*. The implementation of *MathLink* protocol will be presented trough an example of the *SMTIData* function.

The *SMTIData* function is implemented in C as follows:

```
const char *IDataKey[IData_Last] = {"IDataLength",
"RDataLength", "IDataLast", "RDataLast", "LastIntCode",
"Iteration", "TotalIteration", "LinearEstimate", "ErrorStatus",
"MaterialState", "NoSensParameters", "ElementShape",
"SensIndex", "OutputFile", "MissingSubroutine",
"SubDivergence", "ElementState", "NoNodes", "NoElements",
"NoESpec", "Debug", "NoDimensions", "SymmetricTangent",
"NoTmpStore", "NoEquations", "DiagonalSign", "Task",
"NoSubIterations", "CurrentElement", "MaxPhysicalState",
"ExtrapolationType"};

void SMTIData()
{
  int i;
  char *code;

  if(!CheckLinkIntegrity("SMTIData"))return;
  MLGetString(stdlink,&code);

  for(i=0;i<IData[ID_IDataLast];i++){
    if(strcmp(IDataKey[i],code)==0){
      MLFlush(stdlink);
      if(MLReady(stdlink)) MLGetInteger(stdlink,&IData[i]);
      MLPutInteger(stdlink,IData[i]);
      MLDisownString(stdlink,code);
      return;
    }
  };
  MLDisownString(stdlink,code);
  SMTAbort("wrong key","SMTIData",code,0);
  MLPutInteger(stdlink,0);
}
```

The *MathLink* library functions are those starting with ML. In addition *MathLink* requires the template file. In a template file each entry defines a *Mathematica* function that when evaluated calls an associated C function from the interface file. The argument block of the function as it appears in *Mathematica* is also defined in the template file. In the *SMTIData* case the template entry is as follows:

```
:Begin:
:Function:     SMTIData
:Pattern:      SMTIData[i_String]
:Arguments:      {i}
:ArgumentTypes: {Manual}
:ReturnType:    Manual
:End:
```

The template file is processed by Mathematica's utility *mprep* (*MathLink* source file preprocessor), which adds the remote procedure call mechanism part of the C code to the actual function. The resulting C source file is ready for compilation.
Once the code is compiled the *CDriver* is capable of communication with *Mathematica* and its functions are available within *Mathematica.*
For example the *Computational Templates SMTIData* functions are defined in *Mathematica*, using the *CDriver SMTIData* function as follows:

```
SMTIData[i_String] :=
 ExternalCall[LinkObject[C:\Program Files\Wolfram Research\
     Mathematica\4.1\AddOns\Applications\ComputationalTemplates\
     Include/CDriver/ConsoleDriver.exe, 3, 3], CallPacket[3, {i}]]

SMTIData[i_String, j_Integer] :=
 ExternalCall[LinkObject[C:\Program Files\Wolfram Research\
     Mathematica\4.1\AddOns\Applications\ComputationalTemplates\
     Include/CDriver/ConsoleDriver.exe, 3, 3], CallPacket[4, {i, j}]]

SMTIData[i_Integer] :=
 ExternalCall[LinkObject[C:\Program Files\Wolfram Research\
     Mathematica\4.1\AddOns\Applications\ComputationalTemplates\
     Include/CDriver/ConsoleDriver.exe, 3, 3], CallPacket[5, {i}]]

SMTIData[i_Integer, j_Integer] :=
 ExternalCall[LinkObject[C:\Program Files\Wolfram Research\
     Mathematica\4.1\AddOns\Applications\ComputationalTemplates\
     Include/CDriver/ConsoleDriver.exe, 3, 3], CallPacket[6, {i, j}]]

SMTIData[i_, True] := SMTIData[i, 1]

SMTIData[i_, False] := SMTIData[i, 0]
```

If during the *Computational Templates* session the *CDriver* is chosen as a solution environment then all the commands are processed using its C equivalent provided in *CDriver*. The following functions are available in *CDriver*:

- Pre and post processing
  - SMTStructure
  - SMTPost
- Analysis functions
  - SMTNewtonIteration
  - SMTNextStep
  - SMTBackStep
- Data exchange function
  - SMTIData
  - SMTRdata
  - SMTElementData

- SMTSpecData
- SMTNodeData

The question of portability should also be addressed at this point. Current implementation is system dependent due to use to dynamic linked libraries, which are only used on Microsoft Windows systems. The segment of the code related to library loading is very short and can be replaced using the shared libraries available on UNIX systems. The code for loading element routines from shared libraries, as presented in section 3.3.6.2.2, is very similar since the concepts of DLL and shared library are alike. Since there were no requests for UNIX versions the shared library version is not yet implemented as a standard feature. The Mathematica system is available for a wide range of platforms as well as the *MathLink* library and hence it does not impose any additional restrictions on portability of the system.

3.3.6.2 CDriver – data structure implementation

3.3.6.2.1 Environmental Data

The implementation of the environmental data vectors is trivial since both *IData* and *RData* are vectors. Each entry can be accessed using predefined constants using the same naming convention as presented in Table 4 and Table 5.

3.3.6.2.2 ElementSpec

ElementSpec is the most complex data structure appearing in *CDriver* since it contains both data fields and functions responsible for evaluation of element level characteristic quantities. Therefore *ElementSpec* behaves more like an object than a classic C structure although it does not support inheritance. The functions are accessed through function pointers, which are set during the initialization.
For efficiency reason (performance, memory segmentation) the memory is not allocated for each entry in the structure separately. First, the amount of required memory is calculated which is then allocated in two parts. The first part is allocated for integer values and the second one for double values. Pointers are then set to the proper position. This procedure is used for allocation of all basic data types.
Structure *ElementSpec* is defined as follows:

```
typedef struct{
    char *Code;
    struct{
        void (*SetElSpec)(void *);
        void (*SKR)(void *);
```

```
        void (*SRE)(void *);
        void (*SSE)(void *);
        void (*SHI)(void *);
        void (*SPP)(void *);
        void (*User1)(void *);
        void (*User2)(void *);
    }user;
    struct{
        int SpecIndex;
        int NoDimensions;
        int NoDOFGlobal;
        int NoDOFCondense;
        int NoNodes;
        int NoGroupData;
        int NoSegmentPoints;
        int IntCode;
        int NoTimeStorage;
        int NoElementData;
        int NoIntPoints;
        int NoGPostData;
        int NoNPostData;
        int SymmetricTangent;
        int NoIntPointsA;
        int NoIntPointsB;
        int NoIntPointsC;
    }id;
    char **GroupDataNames;
    char **GPostNames;
    char **NPostNames;
    char *Topology;
    int *Segments;
    int *DOFGlobal;
    int *SensType;
    int *SensTypeIndex;
    double *Data;
    double *IntPoints;
} ElementSpec;
```

The names of the entries appearing in the *ElemenSpec* structures are in accordance with descriptions given in Table 6. Due to its importance the element function implementation will be discussed in greater detail.

The element source files generated by the *AceGen* system contain the user subroutines for evaluation of the element level quantities such as element tangent and residual. In order to access these functions from *CDriver* the source files, generated by AceGen, have to be compiled and linked into dynamic linked libraries (element DLL). The advantage of a DLL usage is that the functions are loaded only when needed and they can be updated without having to recompile the entire application.

When the element DLL is loaded into memory other applications can access its functions. Through the *user* function pointers the *CDriver* gets access to the users subroutines from the relevant element DLL. In order to access the proper functions the relevant element DLL have to be loaded and the *user* function pointers have to be

set correctly. In *CDriver* the name of the element DLL is specified by its *Code* as input to the program. The *CDriver* tries to load the element DLL from the disk into the memory. In the case when the element DLL cannot be found it searches for the element source code file and it tries to build the DLL from it.
When the element DLL is loaded the initialization routine checks whether it contains the *SetElSpec* function since this function is essential for proper initialization. The *SetElSpec* function is called first in order to initialize the entire *ElementSpec* data structure. The *user* function pointers are set to the proper addresses of the corresponding functions in DLL. The error code is generated and the *CDriver* execution is terminated if the *SetElSpec* function is not available.
A typical *SetElSpec* function, which initializes the *ElementSpec* structure appears as, follows:

```
void SMTSetElSpec(ElementSpec *es,int *idata,int *ic,double  *gd)
  { static int pn[5]={1, 2, 3, 4, 0};
    static int dof[4]={2, 2, 2, 2};
    static char *gdcs[]={"Elastic modulus (E)",
          "Poisson ratio (ni)","Uniaxial yield stress (Sv)"};
    static char *gpcs[]={"Sxx","Syy","Szz","Sxy","Exx","Eyy",
    "Ezz","Exy","Stress max.","Stress min.","Strain max.",
    "Strain min.","State 0->elastic", "E11-plastic",
   "E22-plastic","E33-plastic","E12-plastic",
   "Plastic multiplier","X","Y","Z","N1",
   "N2","N3","N4"};
    static char *npcs[]={"u - Displacement",
    "v - Displacement"};
   es->Code="Mech4";
   es->id.NoDimensions=2;es->id.NoDOFGlobal=8;
   es->id.NoDOFCondense=0;es->id.NoNodes=4;
   es->id.NoGroupData=3;es->id.NoSegmentPoints=5;
   es->id.IntCode=*ic;es->id.NoElementData=0;
   es->Segments=pn;es->DOFGlobal=dof;
   es->Data=gd;es->id.NoGPostData=25;
   es->id.NoNPostData=2;es->id.SymmetricTangent=1;
   es->IntPoints=SMTMultiIntPoints(ic,idata,
       &es->id.NoIntPoints,&es->id.NoIntPointsA,
       &es->id.NoIntPointsB,&es->id.NoIntPointsC);
   es->id.NoTimeStorage=es->id.NoIntPoints*
       (6+5*idata[ID_NoSensParameters]);
   es->Topology="Q1";es->GroupDataNames=gdcs;
   es->GPostNames=gpcs;es->NPostNames=npcs;
   es->user.SPP=SPP;es->user.SHI=SHI;
   es->user.SSE=SSE;es->user.SKR=SKR;};
```

In the structure the *user* function pointers are defined as generic pointers to functions, which are cast to the actual functions.

The argument block accepted by the *user* functions is as follows

- *v* working field vector

  Element data
- *es* pointer to the element specification (*ElementSpec*)
- *ed* pointer to element data (*ElementData*)
- *nd* pointer to the list of element nodes (*NodeData*)

  Enviroment data
- *rdata* pointer to system data of type double
- *idata* pointer to system data of type integer

  Return data
- *p* pointer to storage of element gradient
- *s* pointer to storage of element Hessian

The working field vector provides the necessary memory space for the function's temporary storage. It is allocated once and it is used for each function call. The pointers supplied in the element data block provide all the required information about the current element. Information about the current state of the analysis is provided in the environment data block while the output of the routine is stored in the return data block. The element and environment blocks are the same for all element level functions while the return data block differs between functions.
SKR is an element function for the evaluation of the residual vector and tangent matrix for current values of the element and node data. The results are stored in vector *p* (residual) and in matrix *s* (tangent). The *SKR* function is called once per element in the iteration step.

```
void SKR(double v[4309],ElementSpec *es,
  ElementData *ed,NodeData **nd,
  double *rdata,int *idata,double *p,
  double **s);
```

Evaluation of post-processing quantities is performed using the *SPP* function. The post-processing quantities are stored according to the position where they are evaluated. The *gpost* list holds the quantities, which are evaluated at the Gauss points while *npost* stores the quantities derived at the nodal points. The return data block therefore consists of the *gpost* and *npost* lists.

```
void SPP(double v[4309],ElementSpec *es,
  ElementData *ed,NodeData **nd,
  double *rdata,int *idata,
  double **gpost,double **npost);
```

3.3.6.2.3 Element Data

The *ElementData* structure is defined, as follows:

```
typedef struct{
    struct{
        int ElemIndex;
        int SpecIndex;
        int Active;
    }id;
    double *Data;
    int *Nodes;
    double *ht;
    double *hp;
} ElementData;
```

The details regarding each field are already provided in Table 7.

3.3.6.2.4 Node data structure

The C definition of the *NodeData* structure is as follows:

```
typedef struct{
     struct {
             int NodeIndex;
             int NoDOF;
             int Fictive;
                } id;
     int *DOF;
     double *X;
     double *Boundary;
     double *at;
     double *ap;
     double *da;
     double *st;
     double *sp;
     double *tmp;
} NodeData;
```

Table 8 contains the detailed description of structure entries.

References

[1] Ahmed K. Noor, *Computational structures technology: leap frogging into the twenty-first century*, Computers & Structures, Vol 73, p.p. 1-31, 1999.

[2] J. Korelc, *Symbolic Approach in Computational Mechanics and its Application to the Enhanced Strain Method*, Ph. D. thesis, Technische Hochschule Darmstadt, 1996.

[3] J. Korelc, *Automatic generation of finite-element code by simultaneous optimalization of expressions*, Theoretical Computer Science, 187, p.p. 231-248, 1997.

[4] J.Korelc, P. Wriggers, *Symbolic approach in computational mechanics*, in Computational Plasticity Fundamentals and Applications (ed. D.R.J. Owen, E. Onate and E. Hinton), pp. 286-304, CIMNE, Barcelona, 1997.

[5] Wolfram, S., *The Mathematica book*, Cambridge University Press, 1996.

[6] J. Korelc, *Automatic generation of numerical codes with introduction to AceGen 4.0 symbolic code generator*, User manual, www.fgg.uni-lj.si/Symech/, 2000

[7] J. Korelc, *Computational Templates*, User manual, www.fgg.uni-lj.si/Symech/, 2000

[8] CIMNE, *GiD pre and post processor*, http://gid.cimne.upc.es/

4 IMPLICIT SOLUTION METHODS FOR NON-LINEAR SYSTEMS

4.1 General formulation

In the following subsections the derivation of tangent operators will be presented for non-linear, transient and coupled problems [1]-[3]. The high abstract level of the presented formulation allows straightforward finite element implementation using the symbolic system [6][7].
All of the problems will be represented by a set of equations in the residual form, which are solved iteratively using the Newton-Raphson method.
Starting from the set of N equations [4]:

$$\Psi_i(a_1,a_2,\ldots,a_N)=0 \qquad i=1,2,\ldots N \tag{4.1}$$

where a_i are the variables and Ψ_i are corresponding functions. Using the matrix notation $\mathbf{a}$ represents the entire vector of values a_i and $\boldsymbol{\Psi}$ denotes a vector of functions Ψ_i. Expanding the functions $\boldsymbol{\Psi}$ in a Taylor series in the neighborhood of $\mathbf{a}$ one can obtain:

$$\boldsymbol{\Psi}(\mathbf{a}+\delta\mathbf{a})=\boldsymbol{\Psi}(\mathbf{a})+\frac{\partial\boldsymbol{\Psi}}{\partial\mathbf{a}}\cdot\delta\mathbf{a}+O(\delta\mathbf{a}^2) \tag{4.2}$$

Neglecting the higher order terms $(\delta\mathbf{a}^2)$ and setting $\boldsymbol{\Psi}(\mathbf{a}+\delta\mathbf{a})=0$ the following set of equations is obtained for the correction $\delta\mathbf{a}$:

$$\frac{\partial\boldsymbol{\Psi}}{\partial\mathbf{a}}\cdot\delta\mathbf{a}=-\boldsymbol{\Psi}(\mathbf{a}) \tag{4.3}$$

Solving the system of equations (4.3) for $\delta\mathbf{a}$ the solution vector can be updated, as:

$$\mathbf{a}_{new}=\mathbf{a}_{old}+\delta\mathbf{a} \tag{4.4}$$

and the process is iterated until convergence is reached.

4.2 Steady state non-linear systems

The steady state non-linear system can be expressed in the residual form as[1]-[3]:

$$\boldsymbol{\Psi}(\mathbf{a})=0 \tag{4.5}$$

where $\mathbf{a}$ is the system response and $\boldsymbol{\Psi}$ is the residual. Using the Newton-Raphson procedure the following iteration is performed:

$$\frac{d\boldsymbol{\Psi}}{d\mathbf{a}}(\mathbf{a}^m)\delta\mathbf{a}^m=-\boldsymbol{\Psi}(\mathbf{a}^m) \tag{4.6}$$

$$\mathbf{a}^{m+1}=\mathbf{a}^m+\delta\mathbf{a}^m \tag{4.7}$$

where $\mathbf{a}^m$ refers to the solution for the current iteration and $\delta\mathbf{a}^m$ the change in the current step increment. Operator $d\boldsymbol{\Psi}/d\mathbf{a}$ is a tangent operator. The procedure of updating the system response $\mathbf{a}$ is repeated until convergence is achieved.

4.3 Transient non-linear systems

In the case of transient problems the response $\mathbf{a}$ is time dependent as well as residual $\boldsymbol{\Psi}$ which is additionally a function of response time derivatives $\dot{\mathbf{a}}$. In the case of numerical solution procedures time derivatives are often approximated using finite differences as follows[1]-[3]:

$${}^n\dot{\mathbf{a}} \approx \frac{{}^n\mathbf{a}-{}^{n-1}\mathbf{a}}{{}^nt-{}^{n-1}t} \tag{4.8}$$

where the index *n-1* refers to the previous time step and *n* to the current time step and *t* is time. The residual (4.5) should be written as:

$${}^n\boldsymbol{\Psi}\left({}^n\mathbf{a},{}^{n-1}\mathbf{a}\right)=\mathbf{0} \tag{4.9}$$

The Newton-Raphson iterative procedure can be utilized and the following equation for incremental response $\delta\mathbf{a}$ can be obtained:

$$\frac{d^n\boldsymbol{\Psi}}{d^n\mathbf{a}}\left({}^n\mathbf{a}^m\right)\delta\mathbf{a}^m = -{}^n\boldsymbol{\Psi}\left({}^n\mathbf{a}^m\right), \tag{4.10}$$

$${}^n\mathbf{a}^{m+1} = {}^n\mathbf{a}^m + \delta\mathbf{a}^m. \tag{4.11}$$

4.4 Steady state coupled non-linear system

Residual for the steady state non-linear system with two distinctive response fields **a** and **b** is of the following form[1]-[3]:

$$\boldsymbol{\Psi}(\mathbf{a},\mathbf{b})=0 \tag{4.12}$$

$$\boldsymbol{\Phi}(\mathbf{a},\mathbf{b})=0 \tag{4.13}$$

where both residuals Ψ and Φ have to be satisfied simultaneously. There are two possibilities regarding the solution of the coupled system. The first option is to form the global residual by assembling (4.12) and (4.13) as

$$\boldsymbol{\Psi}(U)=\begin{bmatrix}\boldsymbol{\Psi}(\mathbf{a},\mathbf{b})\\ \boldsymbol{\Phi}(\mathbf{a},\mathbf{b})\end{bmatrix}=0 \tag{4.14}$$

where

$$U=\begin{bmatrix}\mathbf{a}\\ \mathbf{b}\end{bmatrix} \tag{4.15}$$

Second possibility is to uncouple the system response in such a manner that response **b** is expressed as a function of the system response **a**. The residual (4.12) and (4.13) can be rewritten as:

$$\boldsymbol{\Psi}(\mathbf{a},\mathbf{b}(\mathbf{a}))=0 \tag{4.16}$$

$$\boldsymbol{\Phi}(\mathbf{a},\mathbf{b})=0 \tag{4.17}$$

Solution of the system, consisting of (4.16) and (4.17) is then obtained using a Newton-Raphson iterative procedure in two nested loops. Linearization of (4.16) and (4.17) yields:

$$\left[\frac{\partial \boldsymbol{\Psi}}{\partial \mathbf{a}}(\mathbf{a}^I,\mathbf{b}(\mathbf{a}^I))+\frac{\partial \boldsymbol{\Psi}}{\partial \mathbf{b}}(\mathbf{a}^I,\mathbf{b}(\mathbf{a}^I))\frac{d\mathbf{b}}{d\mathbf{a}}\right]\delta\mathbf{a}^I=-\boldsymbol{\Psi}(\mathbf{a}^I,\mathbf{b}(\mathbf{a}^I)) \tag{4.18}$$

$$\frac{\partial \boldsymbol{\Phi}}{\partial \mathbf{b}}(\mathbf{a}^I,\mathbf{b}^J)\delta\mathbf{b}^J=-\boldsymbol{\Phi}(\mathbf{a}^I,\mathbf{b}^J) \tag{4.19}$$

In the inner loop equation (4.19) is iteratively solved for the incremental response $\delta\mathbf{b}^J$ while current iterate $\mathbf{a}^I$ is fixed. The next iterate $\mathbf{b}^{J+1}$ is evaluated from the incremental response $\delta\mathbf{b}^J$ according to:

$$\mathbf{b}^{J+1}=\mathbf{b}^J+\delta\mathbf{b}^j \tag{4.20}$$

The derivative $(d\mathbf{b}/d\mathbf{a})$ can also be evaluated in the inner loop once $\mathbf{b}(\mathbf{a}^I)$ is known. Differentiating equation (4.17) yields:

$$\frac{\partial \boldsymbol{\Phi}}{\partial \mathbf{a}}(\mathbf{a}^I,\mathbf{b})+\frac{\partial \boldsymbol{\Phi}}{\partial \mathbf{b}}(\mathbf{a}^I,\mathbf{b})\frac{d\mathbf{b}}{d\mathbf{a}}(\mathbf{a}^I)=0 \tag{4.21}$$

and hence the derivative $(d\mathbf{b}/d\mathbf{a})$ is evaluated from

$$\frac{d\mathbf{b}}{d\mathbf{a}}=-\left(\frac{\partial \boldsymbol{\Phi}}{\partial \mathbf{b}}(\mathbf{a}^I,\mathbf{b})\right)^{-1}\frac{\partial \boldsymbol{\Phi}}{\partial \mathbf{a}}(\mathbf{a}^I,\mathbf{b}) \tag{4.22}$$

The derivative $(\partial\boldsymbol{\Phi}/\partial\mathbf{b})(\mathbf{a}^I,\mathbf{b})$ has already been decomposed in evaluation of incremental solution for $\delta\mathbf{b}$ and therefore only back substitution steps are required for evaluation of derivative $(d\mathbf{b}/d\mathbf{a})$.

When the response $\mathbf{b}(\mathbf{a}^I)$ and $(d\mathbf{b}/d\mathbf{a})$ are known the outer loop can be evaluated for the incremental response $\delta\mathbf{a}^I$ according to equation (4.18). Inserting equation (4.22) into (4.18) leads to

$$\left[\frac{\partial \boldsymbol{\Psi}}{\partial \mathbf{a}}(\mathbf{a}^I,\mathbf{b}(\mathbf{a}^I))-\frac{\partial \boldsymbol{\Psi}}{\partial \mathbf{b}}(\mathbf{a}^I,\mathbf{b}(\mathbf{a}^I))\left[\left(\frac{\partial \boldsymbol{\Phi}}{\partial \mathbf{b}}(\mathbf{a}^I,\mathbf{b})\right)^{-1}\frac{\partial \boldsymbol{\Phi}}{\partial \mathbf{a}}(\mathbf{a}^I,\mathbf{b})\right]\right]\delta\mathbf{a}^I \tag{4.23}$$
$$=-\boldsymbol{\Psi}(\mathbf{a}^I,\mathbf{b}(\mathbf{a}^I))$$

System response **a** is then evaluated according to

$$\mathbf{a}^{I+1} = \mathbf{a}^{I} + \delta \mathbf{a}^{I} \tag{4.24}$$

until convergence is achieved.

4.5 Transient coupled non-linear systems

In the case of coupled transient non-linear systems both responses are time dependent and therefore the residuals can be written as follows[1]-[3]:

$${}^{n}\Psi({}^{n}\mathbf{a}, {}^{n-1}\mathbf{a}, {}^{n}\mathbf{b}, {}^{n-1}\mathbf{b}) = 0 \tag{4.25}$$

$${}^{n}\Phi({}^{n}\mathbf{a}, {}^{n-1}\mathbf{a}, {}^{n}\mathbf{b}, {}^{n-1}\mathbf{b}) = 0 \tag{4.26}$$

The responses ${}^{n-1}\mathbf{a}$ and ${}^{n-1}\mathbf{b}$ are known from the previous time step and therefore the equations can be solved for ${}^{n}\mathbf{a}$ and ${}^{n}\mathbf{b}$. The global residual can be formed or the uncoupling procedure can be applied in the same manner as formulated in equations. (4.16) and (4.17). By uncoupling the system presented by (4.25) and (4.26) and dropping known terms from the previous time step (${}^{n-1}$) one can obtain:

$${}^{n}\Psi({}^{n}\mathbf{a}, {}^{n}\mathbf{b}({}^{n}\mathbf{a})) = 0 \tag{4.27}$$

$${}^{n}\Phi({}^{n}\mathbf{a}, {}^{n}\mathbf{b}({}^{n}\mathbf{a})) = 0 \tag{4.28}$$

References

[1] P. Michaleris, D. A. Tortorelli, C. A Vidal, *Tangent Operators and Design Sensitivity Formulations for Transient Non-Linear Coupled Problems with Applications to Elastoplasticity,* Int. Jour. For Numerical Methods in Engineering, vol. 37, pp. 2471-2499, John Wiley & Sons, 1994.

[2] D. A. Tortorelli, *Non-linear and Time Dependent Structural Systems: Sensitivity Analysis and Optimization*, Material for the DCAMM course (held in Lingby, Denmark), Technical University of Denmark, 1997.

[3] D. A. Tortorelli, P. Michaleris, *Design Sensitivity Analysis: Overview and Review,* Inverse Problems in Engineering, vol. 1, pp. 71-105, Overseas Publishers Association, 1994.

[4] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical Recepies in C: The Art of Scientific Computing*, Second Edition, Cambridge University press, 1992.

[5] J. Korelc, *Automatic generation of finite-element code by simultaneous optimalization of expressions*, Theoretical Computer Science, 187, p.p. 231-248, 1997.

[6] J.Korelc, P. Wriggers, *Symbolic approach in computational mechanics*, in Computational Plasticity Fundamentals and Applications (ed. D.R.J. Owen, E. Onate and E. Hinton), pp. 286-304, CIMNE, Barcelona, 1997.

[7] J. Korelc, *Symbolic Approach in Computational Mechanics and its Application to the Enhanced Strain Method*, Ph. D. thesis, Technische Hochschule Darmstadt, 1996.

5 THE FINITE ELEMENT METHOD

In this chapter the solution procedure via the finite element method will be presented since it was used for the numerical solution of all governing equations within the scope of this work. A large amount of literature covers this topic, e.g. references [1]-[7].

In the first part, the formulation of the method is presented and the element level quantities are introduced. Then, the isoparametric family of elements, which was used throughout this work, is discussed. At the end of the chapter numerical integration procedures required for evaluation of the element characteristic quantities are presented as well.

5.1 Finite element formulation

Many physical problems can be described in terms of a set of differential equations of the following form[1]:

$$\mathbf{A}(\mathbf{u})=\begin{Bmatrix} A_1(\mathbf{u}) \\ A_2(\mathbf{u}) \\ \vdots \end{Bmatrix}=0 \tag{5.1}$$

where $\mathbf{u}$ is the unknown function in a domain Ω with the corresponding boundary conditions

$$\mathbf{B}(\mathbf{u})=\begin{Bmatrix} B_1(\mathbf{u}) \\ B_2(\mathbf{u}) \\ \vdots \end{Bmatrix}=0 \tag{5.2}$$

on the boundary Γ. The function $\mathbf{u}$ can be a scalar quantity or a vector of several variables. Additionally, the differential equation may be a single one or a set of

differential equations. As the equation set (5.1) have to be zero at each point in the domain Ω it implies that

$$\int_{\Omega} \mathbf{v}^T \mathbf{A}(\mathbf{u})\, d\Omega = \int_{\Omega} [v_1 A_1(\mathbf{u}) + v_2 A_2(\mathbf{u}) + \cdots]\, d\Omega = 0 \tag{5.3}$$

where $\mathbf{v}$ is a vector of arbitrary functions. If equation (5.3) is satisfied for all $\mathbf{v}$ then the differential equation (5.1) must be satisfied at all points of the domain. Boundary conditions have to be satisfied simultaneously for all functions $\hat{\mathbf{v}}$ requiring

$$\int_{\Gamma} \hat{\mathbf{v}}^T \mathbf{B}(\mathbf{u})\, d\Gamma = \int_{\Gamma} [\hat{v}_1 B_1(\mathbf{u}) + \hat{v}_2 B_2(\mathbf{u}) + \cdots]\, d\Gamma = 0 \tag{5.4}$$

Thus the integral statement equivalent to equations (5.1) and (5.2) can be written

$$\int_{\Omega} \mathbf{v}^T \mathbf{A}(\mathbf{u})\, d\Omega + \int_{\Gamma} \hat{\mathbf{v}}^T \mathbf{B}(\mathbf{u})\, d\Gamma = 0 \tag{5.5}$$

which is satisfied for all $\mathbf{v}$ and $\hat{\mathbf{v}}$. Equation (5.5) imposes certain restrictions on the function $\mathbf{u}$. If the highest order of derivatives occurring in $\mathbf{A}$ or $\mathbf{B}$ is n then the n-1 derivatives has to be continuous (C_{n-1} continuity)[1]. To reduce the required order of continuity of functions $\mathbf{u}$ integration by parts can be performed on (5.5) leading to

$$\int_{\Omega} \mathbf{C}(\mathbf{v})^T \mathbf{D}(\mathbf{u})\, d\Omega + \int_{\Gamma} \mathbf{E}(\hat{\mathbf{v}})^T \mathbf{F}(\mathbf{u})\, d\Gamma = 0 \tag{5.6}$$

where the operators $\mathbf{C}$ and $\mathbf{F}$ contain lower order derivatives than operators $\mathbf{A}$ and $\mathbf{B}$. Due to weakening of continuity restriction the form presented by equation (5.6) is also known as the weak form. The integral statements in (5.5) and (5.6) will be used as a starting point for the finite element approximation.
Using the Galerkin method the unknown function $\mathbf{u}$ is approximated by

$$\mathbf{u} = \sum_{1}^{n} \mathbf{N}_i a_i = \mathbf{N}\mathbf{a} \tag{5.7}$$

and functions $\mathbf{v}$ and $\hat{\mathbf{v}}$ are defined as a finite set of prescribed functions as follows:

$$\mathbf{v} = \mathbf{w}_j \quad \hat{\mathbf{v}} = \hat{\mathbf{w}}_j \tag{5.8}$$

where j ranges from 1 to n.
Introducing approximation (5.7) into (5.5) leads to the following system of algebraic equations

$$\int_{\Omega} \mathbf{w}_j^T \mathbf{A}(\mathbf{Na}) d\Omega + \int_{\Gamma} \hat{\mathbf{w}}_j^T \mathbf{B}(\mathbf{Na}) \, d\Gamma = 0 \tag{5.9}$$

while introduction of (5.7) into (5.6) yields

$$\int_{\Omega} \mathbf{C}(\mathbf{w}_j)^T \mathbf{D}(\mathbf{Na}) \, d\Omega + \int_{\Gamma} \mathbf{E}(\hat{\mathbf{w}}_j)^T \mathbf{F}(\mathbf{Na}) \, d\Gamma = 0 \tag{5.10}$$

Each of equations (5.9) and (5.10) represents a set of algebraic equations which can be solved for unknown parameters **a** using the Newton-Raphson iterative procedure presented in the previous chapter.
Following the Bubnov-Galerkin method, where the shape functions are used as weighting functions, it follows that

$$\mathbf{w}_j = \mathbf{N}_j \tag{5.11}$$

and functions $\hat{\mathbf{w}}_j$ are equal to $\mathbf{N}_j$ evaluated at boundary Γ.

Other common choices are to use the Dirac delta function ($\mathbf{w}_j = \delta_j$ - point collocation method) or the unity matrix ($\mathbf{w}_j = \mathbf{I}$ on subdomain and zero elsewhere - subdomain collocation method).

5.2 Element level description

As discussed in section 5.1, when using the finite element method the domain of the problem Ω is divided into sub-domains called *finite elements* where unknown functions **u** are approximated with trial functions on the sub-domain using the approximation represented by equation (5.7).
The number of unknown parameters **a** is problem dependent while the number of unknown parameters on the single sub-domain remains constant. The goal is therefore to generate a code that evaluates characteristic arrays of a single element while the global arrays (residual and tangent matrix) are assembled numerically[8].
The resulting system of linear equations is also solved numerically.
The evaluation of characteristic arrays includes also integration over the element domain. Rather then using closed form integration, which can be performed only on a few simple linear elements numerical integration is employed. Evaluation of characteristic quantities is hence required only at the integration points. To indicate the value of the characteristic quantities at the integration point an asterisk will be used ($\mathbf{\Psi}^*, \mathbf{K}^*$). Global characteristics are then defined in terms of element characteristics as follows:

$$\mathbf{K}=\sum_{e}\mathbf{K}^{e}$$
$$\mathbf{K}^{e}=\int_{\Omega_e}\mathbf{K}^{*}d\,\Omega_{e} \tag{5.12}$$

$$\mathbf{\Psi}=\sum_{e}\mathbf{\Psi}^{e}$$
$$\mathbf{\Psi}^{e}=\int_{\Gamma_e}\mathbf{\Psi}^{*}d\,\Gamma_{e} \tag{5.13}$$

Different types of elements are available with respect to interpolation of the unknown fields and coordinates. Within the scope of this work isoparametric elements will be used which will be presented in the following subsection.

5.3 Isoparametric elements

Different approaches can be established for mapping between the global coordinate system and reference coordinate system. The mapping relation can be described as follows:

$$\begin{Bmatrix}x\\y\\z\end{Bmatrix}=f\begin{Bmatrix}\xi\\\eta\\\zeta\end{Bmatrix} \tag{5.14}$$

When the unknown functions and geometry is interpolated using the same interpolation functions this is termed the isoparametric approach. The coordinates are interpolated as follows:

$$x=N_1x_1+N_2x_2+\cdots=\mathbf{N}\mathbf{x} \tag{5.15}$$

$$y=N_1y_1+N_2y_2+\cdots=\mathbf{N}\mathbf{y} \tag{5.16}$$

$$z=N_1z_1+N_2z_2+\cdots=\mathbf{N}\mathbf{z} \tag{5.17}$$

where N=N(ξ,η,ζ) are standard shape functions and (x_i,y_i,z_i) are the coordinates of points lying on the element boundary.

The derivatives of the shape functions occurring in the stiffness matrix are with respect to global coordinates while the shape functions are defined in terms of a reference coordinate system (ξ,η,ζ). It is necessary to express the global derivatives in terms of the reference coordinates. Performing the derivation of the shape functions with respect to reference coordinate ξ leads to

$$\frac{\partial N_i}{\partial \xi}=\frac{\partial N_i}{\partial x}\frac{\partial x}{\partial \xi}+\frac{\partial N_i}{\partial y}\frac{\partial y}{\partial \xi}+\frac{\partial N_i}{\partial z}\frac{\partial z}{\partial \xi} \tag{5.18}$$

and hence for the set of reference coordinates

$$\begin{Bmatrix}\frac{\partial N_i}{\partial x}\\ \frac{\partial N_i}{\partial y}\\ \frac{\partial N_i}{\partial z}\end{Bmatrix}=\mathbf{J}^{-1}\begin{Bmatrix}\frac{\partial N_i}{\partial \xi}\\ \frac{\partial N_i}{\partial \eta}\\ \frac{\partial N_i}{\partial \zeta}\end{Bmatrix} \tag{5.19}$$

where J is the Jacobian matrix of the following form

$$\mathbf{J}=\begin{bmatrix}\frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi}\\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta}\\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta}\end{bmatrix} \tag{5.20}$$

To transfer the variables and the integration region to the reference coordinate system the determinant of **J** will be used as follows

$$dx\,dy\,dz=\det\mathbf{J}\,d\xi\,d\eta\,d\zeta \tag{5.21}$$

Integration of the element level quantities is therefore performed in the reference coordinate system maintaining the simple integral limits since the reference variables vary between –1 and 1.
The different topologies of isoparametric elements used within the scope of this work will be presented providing detiled information about nodal points and shape functions.

5.3.1 L1 – Linear element with two nodes

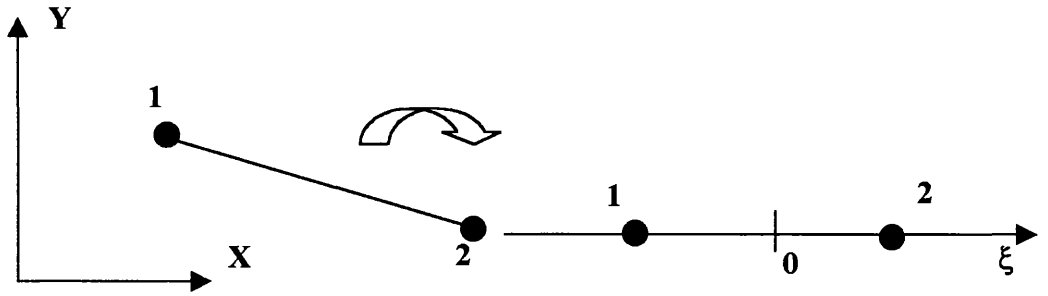


Figure 7 Line element topology in global and reference coordinate system.

Coordinates of nodes in the reference coordinate system are $\xi_1 = -1$ and $\xi_2 = -1$. Standard isoparametric shape functions are defined as follows

$$N_1 = \frac{1}{2}(1-\xi)$$
$$N_2 = \frac{1}{2}(1+\xi) \tag{5.22}$$

5.3.2 T1 - Triangle element with three nodes

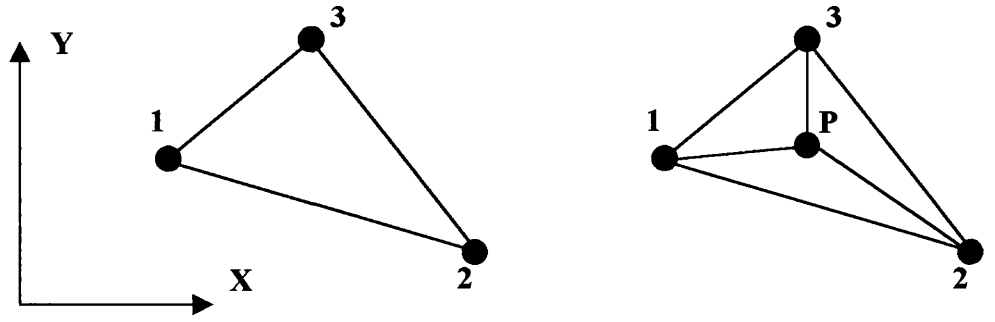


Figure 8 Element topology - triangle with 3 nodes.

The three noded isoparametric triangle element with topology presented in Figure 8 was employed. In the case of triangular elements the reference coordinate system can be defined using area coordinates. Area coordinates (L_1, L_2, L_3) for the point P on Figure 8 are defined as

$$L_1 = \frac{A_{p23}}{A_{123}},\ L_2 = \frac{A_{p13}}{A_{123}}, L_3 = \frac{A_{p12}}{A_{123}} \tag{5.23}$$

where A_{xyz} is the area of the triangle with nodes in points x, y and z.
Positions of nodal points in the reference coordinate system are as follows

Point	L_1	L_2	L_3
1	1	0	0
2	0	1	0
3	0	0	1

The following standard isoparametric shape functions were used

$$\begin{aligned} N_1 &= L_1 \\ N_2 &= L_2 \\ N_3 &= 1 - L_1 - L_2 \end{aligned} \tag{5.24}$$

where N_3 is basically the third area coordinate L_3 which can be expressed as a combination of L_1 and L_2.

5.3.3 Q1 – Quadrilateral with four nodes

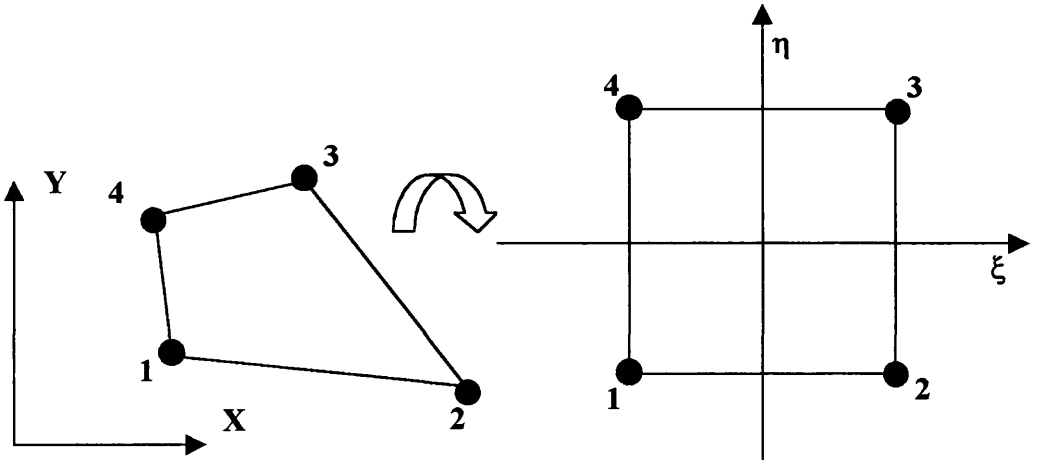


Figure 9 Element topology - quadrilateral with 4 nodes global and reference topology.

Positions of nodal points in the reference coordinate system are as follows

Point	ξ	η
1	-1	-1
2	1	-1
3	1	1
4	-1	1

Isoparametric interpolation functions are given by

$$N_1 = \frac{1}{4}(1-\xi)(1-\eta)$$
$$N_2 = \frac{1}{4}(1+\xi)(1-\eta)$$
$$N_3 = \frac{1}{4}(1+\xi)(1+\eta) \qquad (5.25)$$
$$N_4 = \frac{1}{4}(1-\xi)(1+\eta)$$

5.3.4 Q2 – Quadrilateral with eight nodes

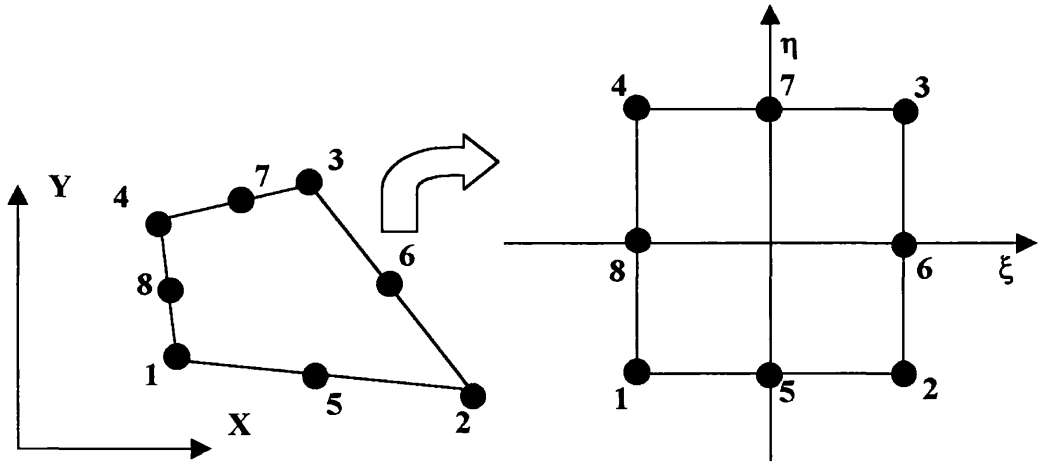


Figure 10 Element topology - quadrilateral with 8 nodes global and reference topology.

Positions of nodal points in the reference coordinate system are as follows

Point	ξ	η
1	-1	-1
2	1	-1
3	1	1
4	-1	1
5	0	-1
6	1	0
7	1	1
8	-1	0

Isoparametric interpolation functions are given by

$$
\begin{aligned}
N_1 &= \frac{1}{4}(1-\xi)(1-\eta)(-\xi-\eta-1) & N_5 &= \frac{1}{2}(1-\xi^2)(1-\eta) \\
N_2 &= \frac{1}{4}(1+\xi)(1-\eta)(\xi-\eta-1) & N_6 &= \frac{1}{2}(1+\xi)(1-\eta^2) \\
N_3 &= \frac{1}{4}(1+\xi)(1+\eta)(\xi+\eta-1) & N_7 &= \frac{1}{2}(1-\xi^2)(1+\eta) \\
N_4 &= \frac{1}{4}(1-\xi)(1+\eta)(-\xi+\eta-1) & N_8 &= \frac{1}{2}(1-\xi)(1-\eta^2)
\end{aligned}
\qquad (5.26)
$$

5.4 Numerical integration

The evaluation of characteristic arrays includes integration over the element domain. Because closed form integration can be performed only on a few simple linear elements numerical integration of the following form is employed

$$\int f\,dA = \sum_{i=1}^{n} w_i\, f(\xi_i, \eta_i) \tag{5.27}$$

where f is any function while (ξ_i, η_i) are the reference coordinates of the i-th integration point and w_i are the corresponding weights.

Details regarding the numerical integration in one and two dimensions are provided in the following subsections together with the reference coordinates of integration points and corresponding weights for different integration rules.

5.4.1 One dimensional numerical integration

Description	No. of points	Disposition
1 point Gauss	1	
2 point Gauss	2	
3 point Gauss	3	
4 point Gauss	4	
5 point Gauss	5	

Table 10 Line element – positions of integration points.

i	ξ_i	η_i	w_i
1	0	0	2
1	-0.57735	0	1
2	0.57735	0	1

1	-0.774597	0	0.555556
2	0	0	0.888889
3	0.774597	0	0.555556
1	-0.861136	0	0.347855
2	-0.339981	0	0.652145
3	0.339981	0	0.652145
4	0.861136	0	0.347855
1	-0.90618	0	0.331101
2	-0.538469	0	0.668876
3	0	0	4.6E-05
4	0.538469	0	0.668876
5	0.90618	0	0.331101

Table 11 Line element – Coordinates and weights of integration points.

5.4.2 Two dimensional numerical integration

5.4.2.1 Numerical integration on triangular topology

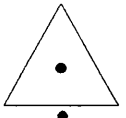
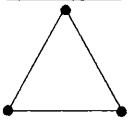
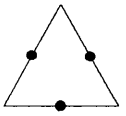
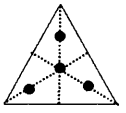
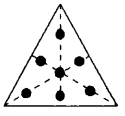
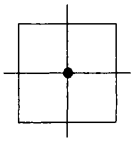
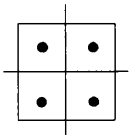
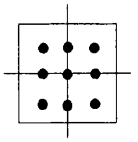
Description	No. of points	Disposition
1 point integration	1	
3 point integration - nodes	3	
3 point integration	3	
4 point integration	4	
7 point integration	7	

Table 12 Triangular elements – Positions of the integration points.

i	ξ_i	η_i	w_i
1	0.333333	0.333333	0.5
1	0	0	0.166667
2	1	0	0.166667
3	0	1	0.166667
1	0.5	0.5	0.166667
2	0	0.5	0.166667
3	0.5	0	0.166667
1	0.333333	0.333333	-0.28125
2	0.6	0.2	0.260417
3	0.2	0.6	0.260417
4	0.2	0.2	0.260417
1	0	0	0.025
2	0.5	0	0.066667
3	1	0	0.025
4	0.5	0.5	0.066667
5	0	1	0.025
6	0	0.5	0.066667
7	0.333333	0.333333	0.225

Table 13 Triangular elements – Coordinates of the integration points and corresponding weights.

5.4.2.2 Numerical integration on quadrilateral topology

Description	No. of points	Disposition
1 point integration	1	
2×2 Gauss integration	4	
3×3 Gauss integration	9	

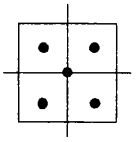
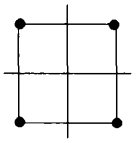
5 point special rule	5	
4 points in nodes	4	

Table 14 Quadrilateral elements – Positions of the integration points.

i	ξ_i	η_i	w_i
1	0	0	4
1	-0.57735	-0.57735	1
2	0.57735	-0.57735	1
3	-0.57735	0.57735	1
4	0.57735	0.57735	1
1	-0.774597	-0.774597	0.308642
2	0	-0.774597	0.493827
3	0.774597	-0.774597	0.308642
4	-0.774597	0	0.493827
5	0	0	0.790123
6	0.774597	0	0.493827
7	-0.774597	0.774597	0.308642
8	0	0.774597	0.493827
9	0.774597	0.774597	0.308642
1	-0.774597	-0.774597	0.555556
2	0.774597	-0.774597	0.555556
3	-0.774597	0.774597	0.555556
4	0.774597	0.774597	0.555556
5	0	0	1.777778
1	-1	-1	1
2	1	-1	1
3	-1	1	1
4	1	1	1

Table 15 Quadrilateral elements – Coordinates of the integration points and corresponding weights.

References

[1] O. C. Zienkiewicz, R. Taylor, *The Finite Element Method*, Vol. 1, 2 (fourth edition), McGraw-Hill, London, 1991.

[2] K.J. Bathe, *Finite Element Procedures*, p.p. 697-745, Prentice Hall, New Jersey, 1996.

[3] J. Bonet, R.D Wood, *Nonlinear Continuum Mechanics for Finite Element Analysis*, Cambridge University Press, Cambridge, 1997.

[4] M. A. Crisfield, *Non-Linear Finite Element Analysis of Solids and Structures*, Vol. 1, John Wiley & Sons, Chichester, 1991.

[5] M. A. Crisfield, *Non-Linear Finite Element Analysis of Solids and Structures*, Vol. 2, John Wiley & Sons, Chichester, 1997.

[6] D. R. J. Owen, E. Hinton, *An Introduction to Finite Element Computations*, Pineridge Press, Swansea, 1979.

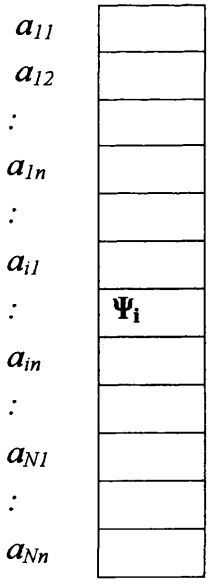
[7] D. R. J. Owen, E. Hinton, *Finite Elements in Plasticity*, Pineridge Press, Swansea, 1980.

[8] J. Korelc, *Symbolic Approach in Computational Mechanics and its Application to the Enhanced Strain Method*, Ph. D. thesis, Technische Hochschule Darmstadt, 1996.

6 FORMULATION OF MAGNETIC, THERMAL AND MECHANICAL PROBLEM

In previous chapters the theoretical aspects, numerical procedures and solution environment for fully coupled non-linear problems were introduced in detail. In this chapter the developed approach will be applied to the derivation of separate computational models for magnetic, thermal and mechanical fields. Individual fields are the required building blocks of a fully coupled model for inductive heating heat treatment process, which will be presented in the next chapter. The subroutines for evaluation of element residual and tangent were generated from symbolic inputs, which will be presented for each element. The element subroutines for evaluation of the residual vector and tangent matrix are based on generated characteristic formulas for evaluation of components of i-th node residual Ψ_i and related stiffness sub-matrix $\mathbf{K}_{ij}$ as presented in Figure 11.

In order to reduce the amount of generated code nodal degrees of freedom can be grouped into subsets. Components of residual Ψ_i and stiffness sub-matrix $\mathbf{K}_{ij}$ belonging to the nodal degrees of freedom from the same subset are evaluated using the same characteristic formula. The number of characteristic formulas is in such case equal to the number of subsets.



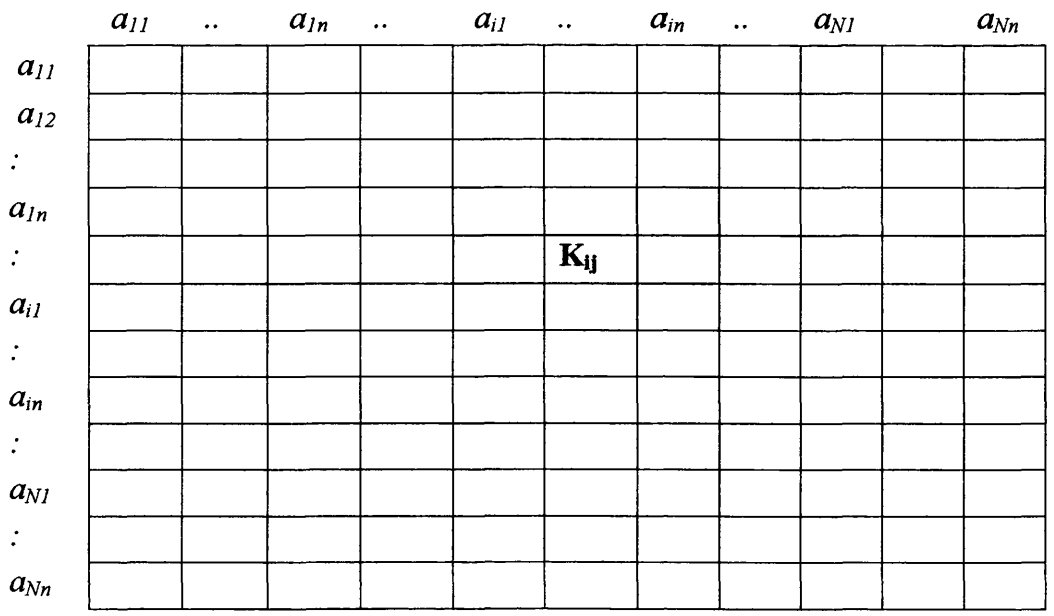


Figure 11 Scheme of element residual Ψ^e and stiffness matrix K^e with respect to nodal residual Ψ_i and tangent sub-matrix K_{ij}.
(a_{ij} is the j-th unknown DOF of the i-th node (i=1 .. N, j=1 .. n)

6.1 Magnetic field

The classical electromagnetic theory starts with the set of Maxwell equations, which as such cannot be derived since they represent mathematical expressions of certain experimental results[6]:

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{6.1}$$

$$\nabla \cdot \mathbf{D} = \rho \tag{6.2}$$

$$\nabla \times \mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t} \tag{6.3}$$

$$\nabla \cdot \mathbf{B} = 0 \tag{6.4}$$

where **H,J,E,D,B** and ρ are, respectively the magnetic field strength, current density, electric field strength, electric flux density, magnetic flux density and charge density. The first equation, better known as Faraday's law, relates the magnetic and electrical fields. The second equation, also known as the Gauss law, shows that flux out of the

closed surface equals the charge enclosed. Ampere's law, represented by the third equation, states that a line integral of magnetic field taken about any given closed path, must equal the current enclosed by this path. Maxwell generalized Ampere's law by including the displacement current term. The fourth equation states that magnetic flux lines must exist as closed lines.
The constitutive relations are:

$$\mathbf{B}=\mu\mathbf{H} \tag{6.5}$$

$$\mathbf{D}=\varepsilon\mathbf{E} \tag{6.6}$$

$$\mathbf{J}=\sigma\mathbf{E} \tag{6.7}$$

where constants μ, ε, and σ are respectively, the magnetic permeability, permitivity and electrical conductivity.
Any vector field can be resolved into an irrotational and a solenoidal part. The irrotational field can be derived from a scalar field called a scalar potential while the solenoidal field can be derived from a vector field called the vector potential. In the case of electromagnetics the use of potentials is preferred for numerical calculations and the following two potentials are commonly introduced[4]-[10]:

ϕ electric scalar potential $\quad \mathbf{E}=-\nabla\phi-\dfrac{\partial\mathbf{A}}{\partial t}$ (6.8)

$\mathbf{A}$ magnetic vector potential $\quad \mathbf{B}=\nabla\times\mathbf{A}$ (6.9)

In terms of the electric scalar and magnetic vector potential and using constitutive relation Eq. (6.6) the first Maxwell equation can be written as follows:

$$\nabla\times\mathbf{H}=\nabla\times(\frac{1}{\mu}\nabla\times\mathbf{A})=-\sigma\nabla\phi-\varepsilon\nabla\left(\frac{\partial\phi}{\partial t}\right)-\sigma\frac{\partial\mathbf{A}}{\partial t}-\varepsilon\frac{\partial^2\mathbf{A}}{\partial^2 t} \tag{6.10}$$

and using the identity $\nabla\times(\frac{1}{\mu}\nabla\times\mathbf{A})=\frac{1}{\mu}\nabla(\nabla\cdot\mathbf{A})-\frac{1}{\mu}\nabla^2\mathbf{A}+\nabla(\frac{1}{\mu})\times\nabla\mathbf{A}$ one can obtain:

$$\frac{1}{\mu}\nabla(\nabla\cdot\mathbf{A})-\frac{1}{\mu}\nabla^2\mathbf{A}+\nabla(\frac{1}{\mu})\times(\nabla\times\mathbf{A})=-\sigma\nabla\phi-\varepsilon\nabla\left(\frac{\partial\phi}{\partial t}\right)-\sigma\frac{\partial\mathbf{A}}{\partial t}-\varepsilon\frac{\partial^2\mathbf{A}}{\partial^2 t} \tag{6.11}$$

One can observe that the introduction of magnetic vector potential **A** (6. 9) does not specify **A** uniquely, because **B** is obtained from **A** by differentiation. To specify **B** in terms of differential operators one must specify the curl and divergence of **A**. Since

the third (6.3) and (6.4) fourth Maxwell equation do not require the divergence of **A** to be specified, one can simply chose:

$$\nabla \cdot \mathbf{A} = 0 \tag{6.12}$$

This is usually called a gauge condition and the particular choice in (6.12) is known as the Coulomb gauge. The other common option is to use the Lorentz gauge:

$$\nabla \cdot \mathbf{A} = -\varepsilon \mu \frac{\partial \phi}{\partial t} \tag{6.13}$$

which is not a common choice for numerical analysis due to implementation difficulties arising from time derivatives[10].
Introducing the Gauss gauge to equation (6.11)leads to:

$$\frac{1}{\mu}\nabla^2 \mathbf{A} - \nabla(\frac{1}{\mu}) \times (\nabla \times \mathbf{A}) = \sigma \nabla \phi + \varepsilon \nabla\left(\frac{\partial \phi}{\partial t}\right) + \sigma \frac{\partial \mathbf{A}}{\partial t} + \varepsilon \frac{\partial^2 \mathbf{A}}{\partial^2 t} \tag{6.14}$$

If a material is a conductor then $\mathbf{J} >> (\partial D/\partial t)$ and then $\sigma >> \omega\varepsilon$ so that we can neglect the displacement current terms[5][7] and equation (6.14) becomes:

$$\frac{1}{\mu}\nabla^2 \mathbf{A} - \nabla(\frac{1}{\mu}) \times (\nabla \times \mathbf{A}) = \sigma \frac{\partial \mathbf{A}}{\partial t} - \mathbf{J}_0 \tag{6.15}$$

where $\mathbf{J}_o = -\sigma \nabla \Phi$ is the source current density.
For the common case where sources of fields vary sinusoidally with time we can write the fields equation in phase notation i.e.:

$$\mathbf{A}(x, y, z, t) = \mathbf{A}(x, y, z) e^{i\omega t} \tag{6.16}$$

$$\mathbf{J}_0(x, y, z, t) = \mathbf{J}_0(x, y, z) e^{i\omega t} \tag{6.17}$$

where multipliers of $e^{i\omega t}$ are called phasors. Introducing the phasor notation we can rewrite Eq.(6.15):

$$\frac{1}{\mu}\nabla^2 \mathbf{A} - \nabla(\frac{1}{\mu}) \times (\nabla \times \mathbf{A}) = i\omega\sigma \mathbf{A} - \mathbf{J}_0 \tag{6.18}$$

Magnetic induction problems are usually rotationally symmetric and therefore the cylindrical coordinates (r, z, θ) can be used. In this case the only nonzero component of the vector potential is $A_\theta(r,z)$. Using cylindrical coordinates and dropping the index θ, equation (6.18) becomes:

$$\frac{1}{\mu}\left[\frac{\partial^2 A}{\partial r^2}+\frac{1}{r}\frac{\partial A}{\partial r}+\frac{\partial^2 A}{\partial z^2}-\frac{A}{r^2}\right]-i\omega\sigma A$$
$$+\frac{\partial(1/\mu)}{\partial r}\left[\frac{1}{r}\frac{\partial A}{\partial r}\right]+\frac{\partial(1/\mu)}{\partial z}\frac{\partial A}{\partial z}+J_0=0 \qquad (6.19)$$

Formulation of a variational problem is required to establish a finite element solution approach. Therefore equation (6.19) can be multiplied by an arbitrary but admissible variation of magnetic vector potential δA.

Performing the multiplication with δA and using the identity $\nabla\cdot(\lambda\mathbf{F})=\lambda\nabla\cdot\mathbf{F}+\mathbf{F}\cdot\nabla\lambda$,where $\mathbf{F}$ is a vector function and λ is a scalar function of space[7], the following equation can be obtained

$$\frac{1}{\mu}\left[\nabla\cdot(\delta A\nabla A)-\nabla A\cdot\nabla\delta A-\frac{A}{r^2}\delta A\right]+$$
$$\left[\frac{\partial(1/\mu)}{\partial r}\left(\frac{1}{r}\frac{\partial r A}{\partial r}\right)+\frac{\partial(1/\mu)}{\partial z}\frac{\partial r A}{\partial z}\right]\delta A+\left[-i\omega\sigma A+J_0\right]\delta A=0 \qquad (6.20)$$

Assuming that the magnetic permeability (μ) is constant within the domain Ω bounded by surface Γ simplifies equation (6.20) by canceling partial derivatives of μ with respect to spatial coordinates r and z. Integrating equation (6.20) over a domain Ω and transforming the first term by means of the divergence theorem, to reduce the order of partial derivatives, yields to the weak form of partial differential equation (6.18) as:

$$\int_\Omega\frac{1}{\mu}\nabla\mathbf{A}\cdot\nabla\delta\mathbf{A}\,d\Omega+\int_\Omega\left(\frac{1}{\mu r^2}+i\omega\sigma\right)\mathbf{A}\,\delta\mathbf{A}\,d\Omega=$$
$$\int_\Omega\mathbf{J}_0\,\delta\mathbf{A}\,d\Omega+\int_\Gamma\frac{1}{\mu}\frac{\partial\mathbf{A}}{\partial n}\delta\mathbf{A}\,d\Gamma \qquad (6.21)$$

Equation (6.21) can be used as a basis for a finite element formulation of electromagnetic induction problems.

6.1.1 Magnetic element implementation

Based on the formulation developed in the previous section (equation (6.21)) finite elements were derived using a symbolic approach. Using *Computational Templates* the symbolic input for axisymmetric three noded triangle, four noded and eight noded quadrilaterals was created. Each element has eight degrees of freedom as

presented in the quadrilateral example in Figure 12. The degrees of freedom are numbered as presented in brackets.

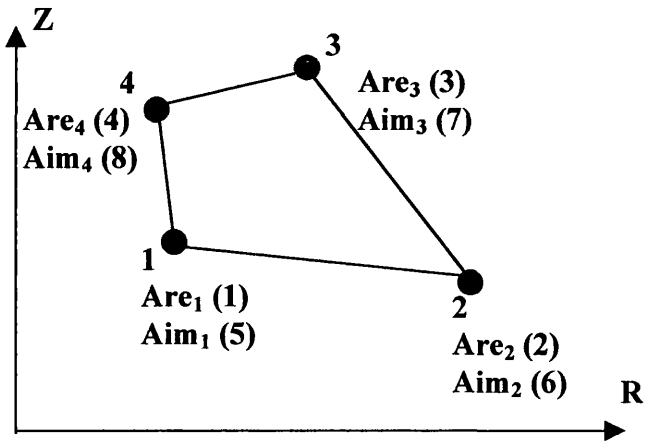


Figure 12 Magnetic quadrilateral element with eight degrees of freedom (two degrees of freedom per node)

Symbolic definition of a magnetic four noded quadrilateral element consist of the following steps:

Step 1: Initialization

- The AceGen and the Compuational templates are initialized and the basic element options are set. The element will have the four noded quadrilateral topology (code Q1) with two global degrees of freedom per integration point (real and imaginary part of the magnetic vector poten tial). The subroutine for the evaluation of the tangent matrix and residual vector will be generated.

```
SMSInitialize["Mag4", "VectorLength" → 1000, "Mode" -> "Prototype",
  "Language" → "C++"];
SMTInitialize["Mag4", "CDriver", "SMTTopology" → "Q1", "SMTDOFGlobal" → 2,
  "SMTSymmetricTangent" → 0];
SMTUserSubroutine["Tangent and residual"];
```

Step 2: Input data interface

- The coordinates of the element nodes and the current values of the global degrees of freedom are supplied to the routine.

```
Ri = Array[SMSReal[nd$$[#, "X", 1], {{-1, 1, 1, -1}[[#]] + SMSRandom[]}] &, 4];
Zi = Array[SMSReal[nd$$[#, "X", 2], {{-1, -1, 1, 1}[[#]] + SMSRandom[]}] &, 4];
ARei = Array[SMSReal[nd$$[#, "at", 1]] &, 4];
AImi = Array[SMSReal[nd$$[#, "at", 2]] &, 4];
```

- Material parameters are supplied.

```
SMTGroupDataNames = {"Permeability μ", "Frequency ω", "Conductivity γ",
   "Source current J"};
{μmg, ωeg, γeg, Jgg} = Array[SMSReal[es$$["Data", #]] &, 4];
```

- Start of the loop over integration points where ξ and η are the local coordinates of the current integration point and *wGauss* is its corresponding weight.

```
NoIp = SMSInteger[es$$["id", "NoIntPoints"]];
SMSDo[IpIndex, 1, NoIp];
{ξ, η, wGauss} = Map[SMSReal[es$$["IntPoints", #1, IpIndex]] &, {1, 2, 4}];
```

Step 3: Definiton of the trial functions

- Definition of the shape functions, interpolation of the physical coordinates and global degrees of freedom. Definition of the Jacobian matrix for the isoparametric mapping from global to local coordinates. Due to axial symmetry about the *y* axis the integration should include multipli cation by $2\,\pi$ r.

```
Ni = 1/4 {(1 - ξ) (1 - η), (1 + ξ) (1 - η), (1 + ξ) (1 + η), (1 - ξ) (1 + η)};
R = SMSFreeze[Ni.Ri];
Z = SMSFreeze[Ni.Zi];
Jm = | SMSD[R, ξ]  SMSD[R, η] |
     | SMSD[Z, ξ]  SMSD[Z, η] |;
Jd = Det[Jm];
SMSDefineDerivative[{ξ, η}, {R, Z}, SMSSimplify[SMSInverse[Jm]]];
SMSDefineDerivative[{R, Z}, {R, Z}, IdentityMatrix[2]];
fGauss = 2 π R Jd wGauss ;
ARe = Ni.ARei;
AIm = Ni.AImi;
```

Step 4: Magnetic equation

- Residual and tangent matrix are generated for a characteristic node and therefore the loop over the nodes is required.

```
SMSDo[i, 1, SMTNoNodes];
```

- Automatic separation of the magnetic weak form equation ΨA to its real ΨArei and imaginary part ΨAimi is performed. The benefit of such separation is the possibility to use a standard solver for the solution of problems with complex degrees of freedom.

```
δARe = SMSD[ARe, ARei, i];
δAIm = SMSD[AIm, AImi, i];
A = ARe + I AIm;
δA = δARe + I δAIm;
ΨA =
  fGauss ( 1/(μmg) ((SMSD[A, {R, Z}].SMSD[δA, {R, Z}])) + (1/(μmg R^2) + I ωeg γeg) A δA -
      Jgg δA);
ΨAi = ComplexExpand[ΨA];
ΨARei = ΨAi /. I → 0;
ΨAImi = ΨAi - ΨARei /. I → 1;
```

- The element residual vector is exported as a routine output.

```
SMSExport[{ΨARei, ΨAImi}, Array[p$$[SMTMaxNoDOFNode (i - 1) + #1] &,
   SMTMaxNoDOFNode], "AddIn" → True];
```

- The element stiffness matrix is exported as a routine output.

```
SMSDo[j, 1, SMTNoNodes];
ARej = SMSPart[ARei, j];
AImj = SMSPart[AImi, j];
KΨ = SMSD[{ΨARei, ΨAImi}, {ARej, AImj}];
SMSExport[KΨ,
 Array[s$$[SMTMaxNoDOFNode (i - 1) + #1, SMTMaxNoDOFNode (j - 1) + #2] &,
  {SMTMaxNoDOFNode, SMTMaxNoDOFNode}], "AddIn" → True];
SMSEndDo[];
```

- The loops over the nodes and integration points are ended

```
SMSEndDo[];
SMSEndDo[]; (* end of the integration loop*)
```

Step 5: Code generation

```
SMSWrite["Mag4", "Splice" → SMTSplice];
```

6.1.2 Verification of the magnetic element

The derived magnetic field computational model was verified using the example presented in Figure 13 where a single circular coil is placed above the half-space. The analytical solution of the problem is available in the literature [5][4].
Values of parameters used in the calculation are presented in Table 16.

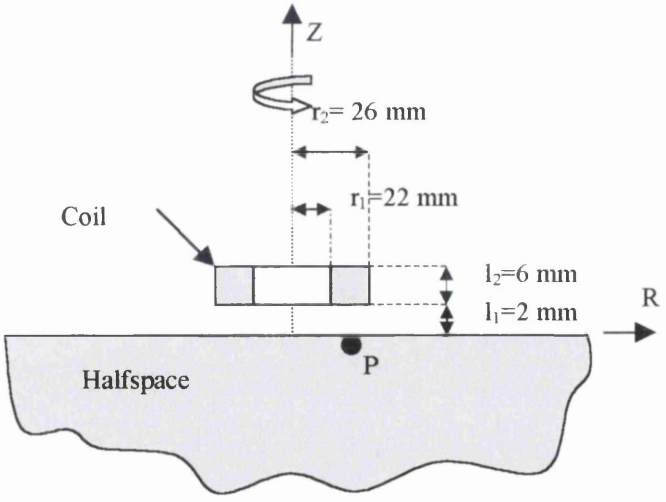


Figure 13 Schematic of the induction heating of a half-space by a single circular coil above it.

The analytical solution for the magnetic vector potential of the half-space example was obtained using Green's functions:

$$A_1(r,z)=\frac{\mu_0 J_g}{2}\int_0^{\infty}\frac{1}{\alpha^3}F(r_1,r_2)J_1(\alpha,r)\left[(e^{\alpha l_2}-e^{\alpha l_1})+w(e^{-\alpha l_1}-e^{-\alpha l_2})\right]e^{-\alpha z}d\alpha \tag{6.22}$$

$$A_2(r,z)=\frac{\mu_0 J_g}{2}\int_0^{\infty}\frac{1}{\alpha^3}F(r_1,r_2)J_1(\alpha,r)(e^{-\alpha l_1}-e^{-\alpha l_2})(e^{\alpha z}+we^{-\alpha z})d\alpha \tag{6.23}$$

$$A_3(r,z)=\frac{\mu_0 J_g}{2}\int_0^{\infty}\frac{1}{\alpha^3}F(r_1,r_2)J_1(\alpha,r)(e^{\alpha l_2}-e^{-\alpha l_1})+(1+w)e^{\alpha_1 z}d\alpha \tag{6.24}$$

$$\begin{aligned}A_{12}(r,z)=\frac{\mu_0 J_g}{2}\int_0^{\infty}\frac{1}{\alpha^3}F(r_1,r_2)J_1(\alpha,r)(&\left[(e^{\alpha z}-e^{-\alpha l_1})+w(e^{-\alpha l}-e^{-\alpha z})\right]e^{-\alpha z}\\&+(e^{-\alpha z}-e^{-\alpha l_2})(e^{\alpha z}+we^{-\alpha z}))d\alpha\end{aligned} \tag{6.25}$$

$$F(r_1,r_2)=\alpha^2\int_{r_1}^{r_2}r_0J_1(\alpha,r_0)dr_0\ ;\ w=\frac{\mu_r\alpha-\alpha_1}{\mu_r\alpha+\alpha_1}\ ;\ \mu_r=\frac{\mu_1}{\mu_0};\alpha_1=\sqrt{\alpha^2+i\omega\mu_1\sigma_1} \tag{6.26}$$

where the index i (A_i) 1 refers to the workpiece, 2 to the coil, 3 to the air and 12 to the air gap between coil and the workpiece. J_0 and J_1 are Bessel functions of the first kind of order zero and one.
Numerical solution was obtained using an unstructured mesh (Figure 14) consisting of 1941 quadrilateral elements (1961 nodes) with two degrees of freedom per node.

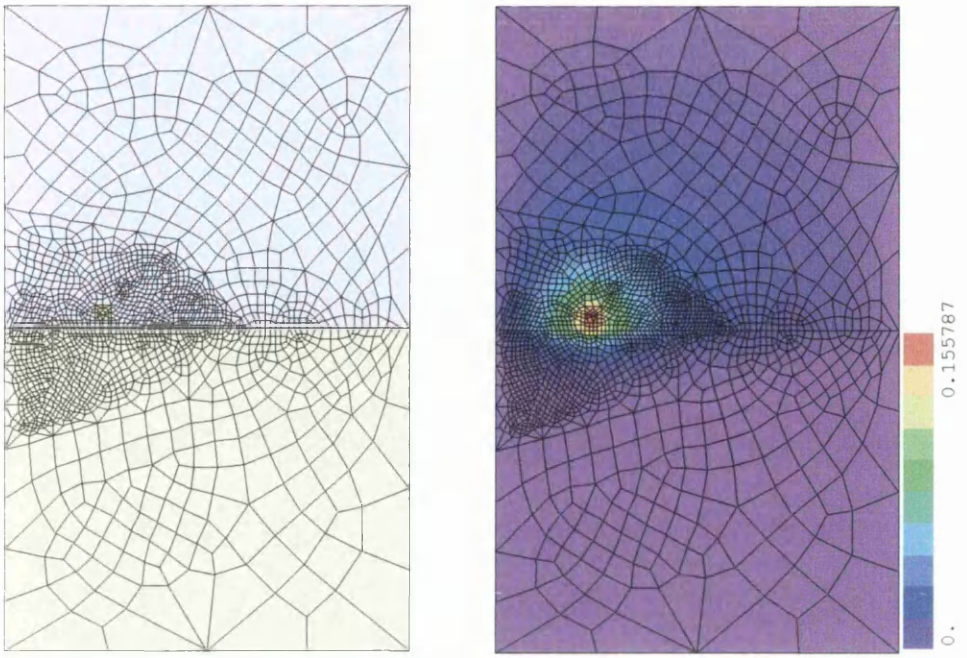


Figure 14 Finite element mesh and results for the absolute value of the magnetic vector potential

Symbol	Description	Unit	Halfspace	Coil	Air
μ	Magnetic permeability	$\frac{H}{m}$	90 μ_0	μ_0	μ_0
ω	Frequency	$\frac{1}{s}$	60	60	60
σ	Electric conductivity	$\frac{1}{\Omega m}$	$3\ 10^6$	$5.7\ 10^7$	0
J_g	Source current density	$\frac{A}{m^2}$	0	$1.2\ 10^{10}$	0

Table 16 Values of material parameters for verification of the magnetic field element ($\mu_0 = 4\ \pi\ 10^{-7}$)

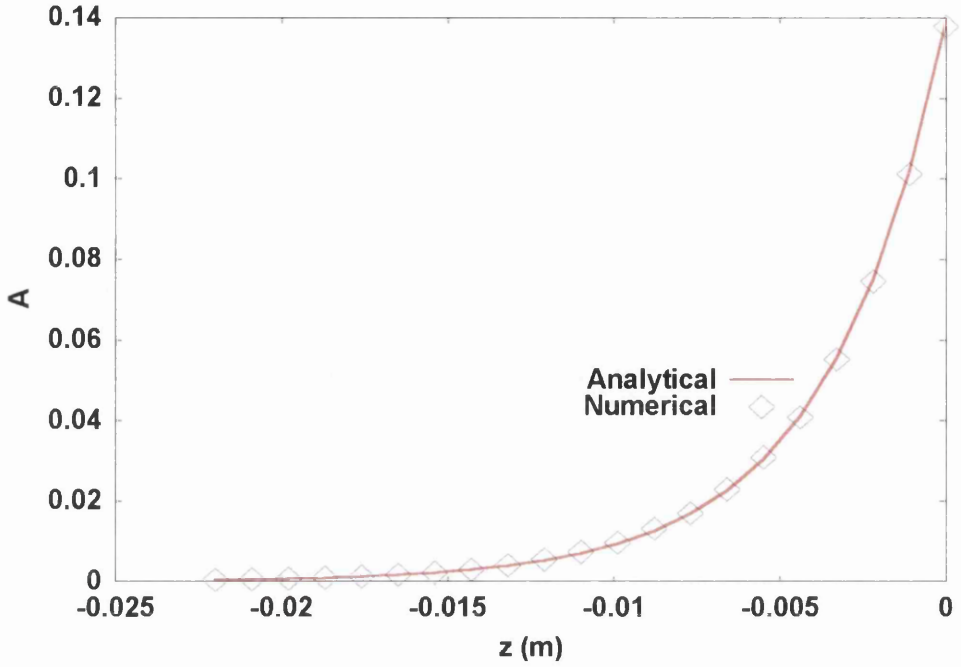


Figure 15 Comparison of numerical and analytical solution for absolute value of the magnetic vector potential in a halfspace (at r=0.024)

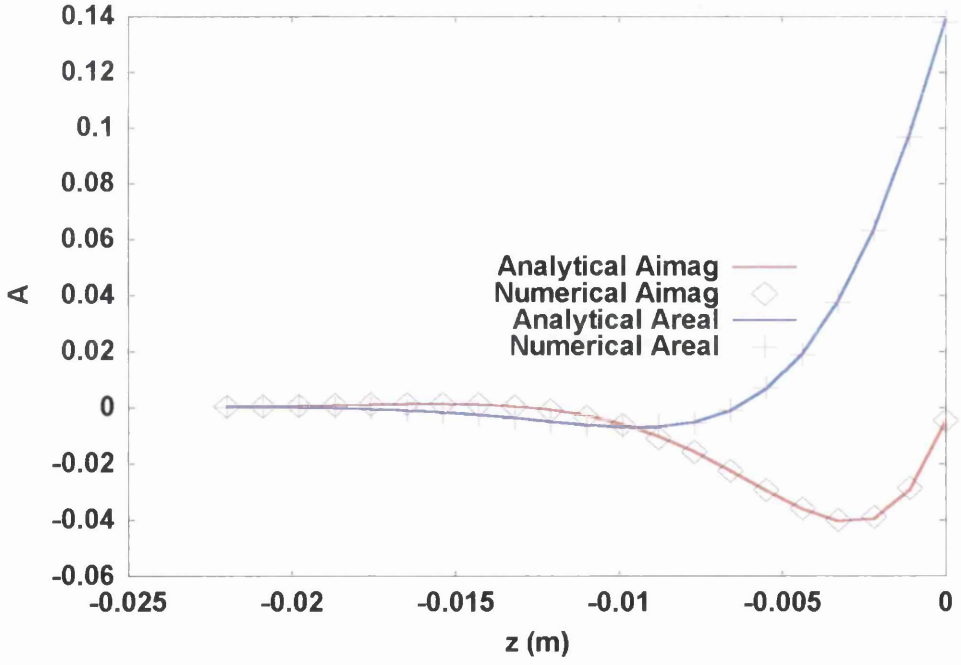


Figure 16 Comparison of numerical and analytical solution for real and imaginary component of the magnetic vector in a halfspace (at r=0.024)

The analytical solution was evaluated in the halfspace along a line parallel to the *z*-axis through point *P* (see Figure 13). It is evident from the results presented in Figure 15 and Figure 16 that there is a good agreement between numerical and analytical results for the magnetic vector potential.

6.2 Thermal field

Conduction of heat in an isotropic solid with internal heat source *q(x,y,z,t)* is described by the following differential equation[11]-[13]:

$$\nabla \cdot (k \nabla T) + q = \rho c_p \frac{\partial T}{\partial t} \tag{6.27}$$

where k is the thermal conductivity, ρ is the density and c_p is the specific heat. A problem in heat conduction for a body occupying a volume Ω with boundary Γ is to find a continuously differentiable function *T(x,y,z,t)* which satisfies equation (6.27) and the following initial and boundary conditions:

initial condition: $$T = T_o(x, z, y, 0) \text{ in } \Omega \text{ and } \Gamma \tag{6.28}$$

boundary conditions: $$T = T_u(x, z, y, t) \text{ on } \Gamma_u \tag{6.29}$$

$$k \frac{\partial T}{\partial n} = q_s \text{ on } \Gamma_s \tag{6.30}$$

where $\frac{\partial T}{\partial n}$ is a directional derivative. T_u and q_s are prescribed values of temperature and heat flux on the boundaries. To obtain the weak form equation (6.27) can be multiplied by a variation of temperature δT and integrated by parts (using Green's theorem) leading to[15]:

$$\int_\Omega \rho c_p \frac{\partial T}{\partial t} \delta T \, d\Omega + \int_\Omega \nabla \delta T \cdot (k \nabla T) d\Omega = \int_\Omega q \, \delta T \, d\Omega + \int_{\Gamma_s} \delta T \, q_s d\Gamma \tag{6.31}$$

Equation (6.31) represents the starting point for the finite element discretization.

6.2.1 Thermal element implementation

As in the case of the magnetic element in Section 6.1.1 the three nodded triangular, four and eight noded axisymmetric elements were derived using symbolic description as presented on Figure 17.

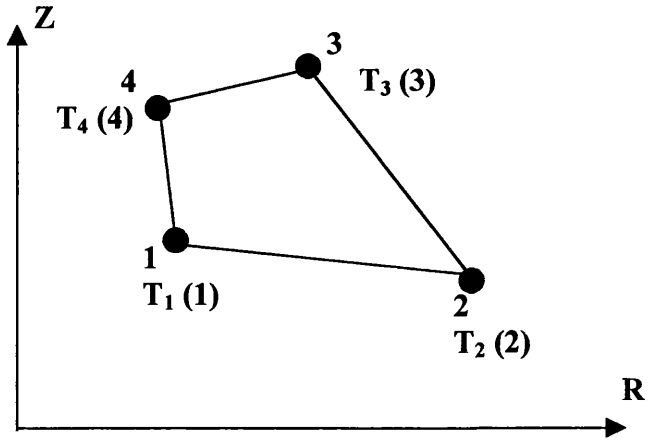


Figure 17 Thermal quadrilateral element with four degrees of freedom (one degree of freedom per node)

The symbolic input for the thermal element consists of the following steps:

Step 1: Initialization

- The AceGen and the Computational templates are initialized and the basic element options are set. The element will have the four noded quadrilateral topology (code Q1) with one global degree of freedom per integration point (temperature).

```
SMSInitialize["Thermo4", "VectorLength" → 1000, "Mode" -> "Prototype",
  "Language" → "C++"];
SMTInitialize["Thermo4", "CDriver", "SMTTopology" → "Q1",
  "SMTDOFGlobal" → 1];
SMTUserSubroutine["Tangent and residual"];
```

Step 2: Input data interface

- Coordinates and current values of temperature are taken from the supplied arguments as well as the current value of the time increment.

```
Ri = Array[SMSReal[nd$$[#, "X", 1], {{-1, 1, 1, -1}[[#]] + SMSRandom[]}] &, 4];
Zi = Array[SMSReal[nd$$[#, "X", 2], {{-1, -1, 1, 1}[[#]] + SMSRandom[]}] &, 4];
ΔT = SMSReal[rdata$$["TimeIncrement"]];
Tti = Array[SMSReal[nd$$[#, "at", 1]] &, 4];
Tpi = Array[SMSReal[nd$$[#, "ap", 1]] &, 4];
```

- Material parameters are supplied.

```
SMTGroupDataNames = {"Density ρ", "Specific heat c",
   "Thermal conductivity k", "Source q"};
{ρg, Chg, khg, qhg} = Array[SMSReal[es$$["Data", #]] &, 4];
```

- Start of the loop over integration points where ξ and η are the local coordinates of the current integration point and *wGauss* is its corresponding weight.

```
NoIp = SMSInteger[es$$["id", "NoIntPoints"]];
SMSDo[IpIndex, 1, NoIp];
{ξ, η, wGauss} = Map[SMSReal[es$$["IntPoints", #1, IpIndex]] &, {1, 2, 4}];
```

Step 3: Definiton of the trial functions

- Definition of the shape functions, interpolation of the physical coordinates and global degrees of freedom. Definition of the Jacobian matrix for the isoparametric mapping from global to local coordinates. Due to axial symmetry about the *y* axis the integration should include multipli cation by $2\,\pi$ r.

```
Ni = 1/4 {(1 - ξ) (1 - η), (1 + ξ) (1 - η), (1 + ξ) (1 + η), (1 - ξ) (1 + η)};
R = SMSFreeze[Ni.Ri];
Z = SMSFreeze[Ni.Zi];
Jm = | SMSD[R, ξ]  SMSD[R, η] |
     | SMSD[Z, ξ]  SMSD[Z, η] |;
Jd = Det[Jm];
SMSDefineDerivative[{ξ, η}, {R, Z}, SMSSimplify[SMSInverse[Jm]]];
SMSDefineDerivative[{R, Z}, {R, Z}, IdentityMatrix[2]];
fGauss = 2 π R Jd wGauss;
Tt = Ni.Tti;
Tp = Ni.Tpi;
```

Step 4: Thermal field equation

- The thermal field equation is evaluated for a characteristic node.

```
SMSDo[i, 1, SMTNoNodes];
```

- Definition of the thermal field weak form equations.

```
δTi = SMSD[Tt, Tti, i];
ΨTi = fGauss ( ρg Chg (Tt - Tp) / ΔT δTi +
     khg SMSD[δTi, {R, Z}].SMSD[Tt, {R, Z}] - qhg δTi);
```

- The element residual vector is exported as a routine output..

```
SMSExport[ΨTi, p$$[i], "AddIn" → True];
```

- The element stiffness matrix is exported as a routine output.

```
    SMSDo[j, 1, SMTNoNodes];
        Ttj = SMSPart[Tti, j];
        KΨ = SMSD[ΨTi, Ttj];
        SMSExport[KΨ, s$$[i, j], "AddIn" → True];
    SMSEndDo[];
```

- The loops over the node and integration loop are ended

```
SMSEndDo[]; (* end loop over the nodes*)
SMSEndDo[]; (* end of the integration loop*)
```

Step 5: Code generation

```
SMSWrite["Thermo4", "Splice" → SMTSplice];
```

6.2.2 Surface flux element

From the implementation of the thermal element it is clear that the natural boundary conditions specified on the element surface are not included in the formulation. In order to implement surface natural boundary conditions a special element has to be generated which will include only boundary related terms. As an example of surface boundary condition implementation the thermal convective boundary condition will be addressed.
The thermal boundary condition presented by equation (6.30) can be rewritten in the case of convection as follows

$$k\frac{\partial T}{\partial n}=h(T)(T_\infty - T_s) \tag{6.32}$$

where T_s and T_∞ are surface and surroundings temperatures while $h(T)$ is the convection coefficient. Including contribution of the convection to the prescribed flux boundary term in equation (6.31) leads to

$$\int_{\Gamma_s}\delta T\, q_s\, d\Gamma = \int_{\Gamma_s}\delta T\, h(T)(T_\infty - T_s)\, d\Gamma \tag{6.33}$$

Based on equation (6.33) the convective surface element can be created. The element can be used also for cases where pseudo-convective boundary conditions are used. In this case the effects of both convection and radiation are described by a single pseudo-convective heat transfer coefficient, which is then multiplied by the temperature difference as in equation (6.33). This approach is quite common in experimental practice due to difficulties related to separation of influences of convection and radiation effects from the experimental results.
The surface flux element is presented in Figure 18.

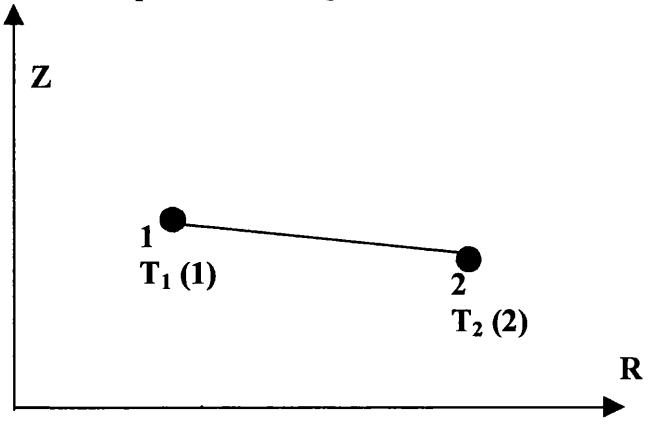


Figure 18 Scheme of two noded surface flux element with one degree of freedom per node.

Symbolic input for such an element consist of the following steps:

Step 1: Initialization

- The AceGen and the Computational templates are initialized and the basic element options are set. In our case the boundary consists of linear segments and therefore the line element will be created (topology code L1) with a single degree of freedom.

```
SMSInitialize["ThermalBound", "Language" → "C++", "Mode" -> "Prototype"];
SMTInitialize["ThermalBound", "CDriver", "SMTTopology" → "L1",
  "SMTNoNodes" → 2,  "SMTDOFGlobal" → 1, "SMTSymmetricTangent" → 0];
SMTUserSubroutine["Tangent and residual"];
```

Step 2: Input data interface

- Coordinates and current values of temperature are taken from the supplied arguments.

```
ri = SMSReal[{nd$$[1, "X", 1], nd$$[1, "X", 2]}];
rj = SMSReal[{nd$$[2, "X", 1], nd$$[2, "X", 2]}];
L = Sqrt[(rj - ri).(rj - ri)];
Ti = Array[SMSReal[nd$$[#, "at", 1]] &, 2];
```

- Pseudoconvective parameters are supplied.

```
SMTGroupDataNames = {"Pseudoconvective coefficient",
   "Temperature of sorround"};
{α, Tinf} = SMSReal[{es$$["Data", 1], es$$["Data", 2]}];
```

Step 3: Definiton of the trial functions

- Definition of the shape functions, interpolation of the physical coordinates and global degrees of freedom.

```
ξ = SMSFictive[];
R = (ri + (1 / 2 + ξ / 2) (rj - ri))[[1]];
T = 1/2 {(1 - ξ), (1 + ξ)}.Ti;
```

Step 4: Pseudo-convective equation

- Definition of the functional. In this simple case closed form integration is posible.

```
δT = SMSD[T, Ti];
Ψi = L π Integrate[R δT α (T - Tinf), {ξ, -1, 1}];
```

- The element residual and stiffness matrix is exported as a routine output.

```
Kij = SMSD[Ψi, Ti];
SMSExport[Ψi, p$$, "AddIn" → True];
SMSExport[Kij, s$$, "AddIn" → True];
```

Step 5: Code generation

```
SMSWrite["ThermalBound", "Splice" → SMTSplice];
```

6.2.3 Verification of the thermal element

Verification of the thermal element was performed on the example presented in Figure 19 where an infinite cylinder is heated by a convective flow and radiation. The effect of both heating mechanisms is described by a pseudo-convective coefficient. The process is transient and hence the distribution of temperature is a function of time and spatial coordinates.
The analytical solution is available in the literature[14] of the following form

$$T(r,t)=T_{\infty}-(T_{\infty}-T_0)\sum_{k=1}^{\infty}\frac{2J_1(\alpha_k)}{\alpha_k(J_0(\alpha_k)^2+J_1(\alpha_k)^2)}Exp\left[-\alpha_k{}^2\frac{\lambda t}{R^2}\right]J_0(\alpha_k\frac{r}{R}) \qquad (6.34)$$

where $\lambda=\frac{k}{c_p\rho}$, J_0 and J_1 are Bessel functions of the first kind of order zero and one, T_0 and T_∞ are the initial and surrounding temperatures and α_k are the roots of the equation

$$\alpha J_1(\alpha)-J_o(\alpha)Bi=0 \qquad (6.35)$$

where $Bi=\frac{hR}{k}$ is the Biot number.
The analytical solution was evaluated with the set of parameters presented in Table 17.

Parameter	Value
ρ	7850 kg/m^3
c_p	711.8 J/kg K
k	40.7 W/m K
h	290.8 W/m^2 K
R	0.5 m
T_0	20 °C
T_∞	950 °C

Table 17 Parameters used in verification calculation

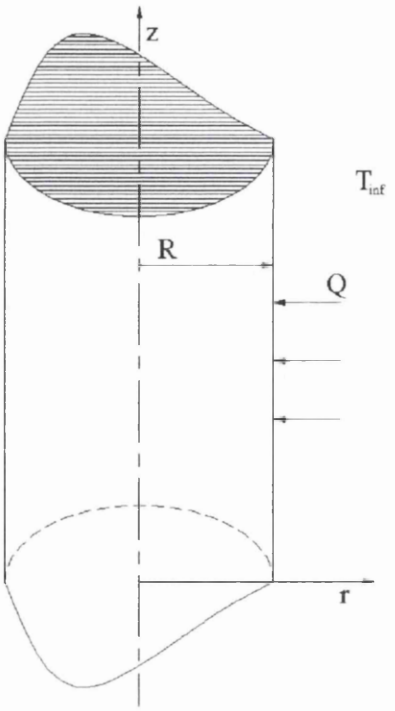


Figure 19 Heating of the infinite cylinder

For finite element evaluation a structured mesh of 40 x 40 quadrilateral elements was used. Boundary conditions were imposed using additional 40 line elements as presented in Figure 20.

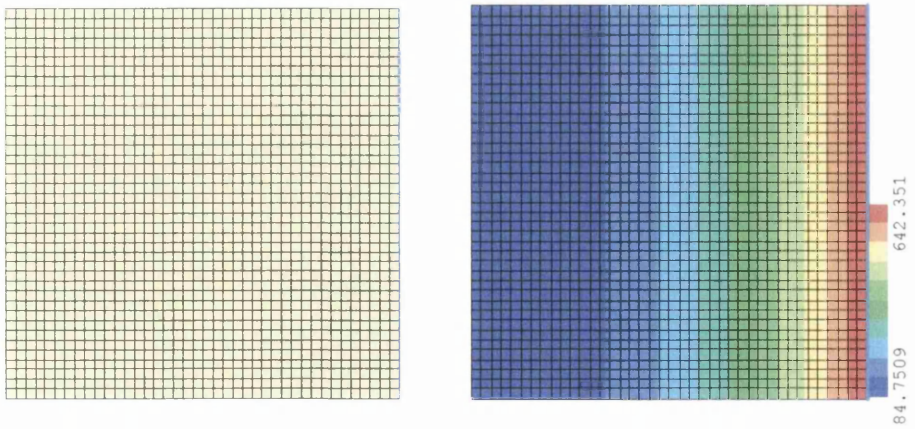


Figure 20 Finite element mesh and temperature distribution

The results of the verification are presented in Figure 21, Figure 22 and Figure 23 where the numerical results are compared with the analytical solution at three different locations.

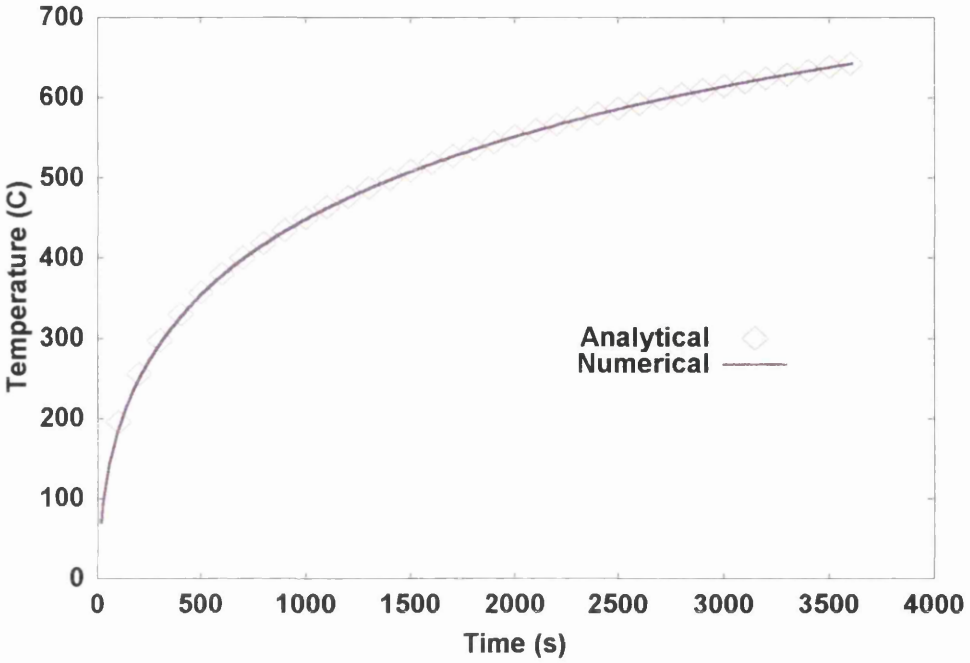


Figure 21 Verification of thermal elements. Temperature evolution at r = R

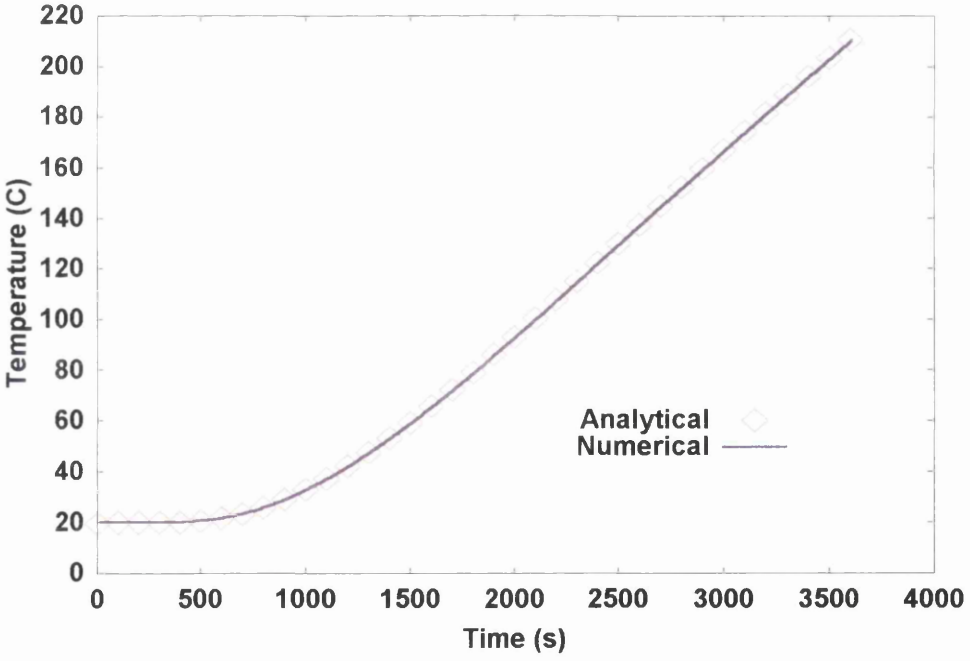


Figure 22 Verification of thermal elements. Temperature evolution at r = 0.5 R

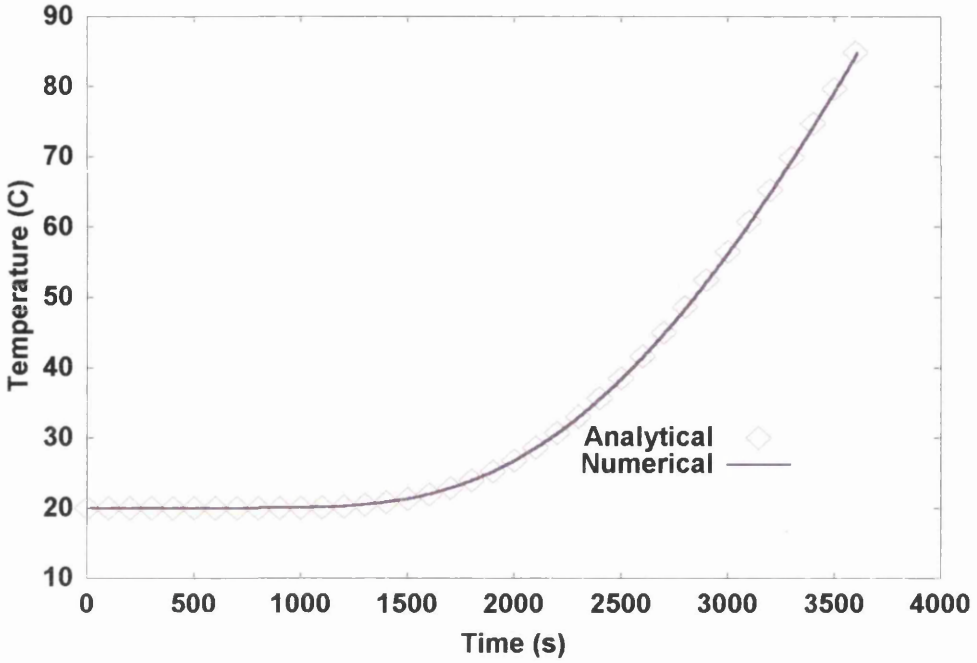


Figure 23 Verification of thermal elements. Temperature evolution at r = 0

The analytical solutions are plotted with the full lines while the numerical solutions are represented as points. The numerical results are in good agreement with the analytical solution.

6.3 Mechanical model

6.3.1 Continuum mechanics

Several comprehensive books covering the field of continuum mechanics are available[2][3] and within the scope of this work only the basic definitions of displacements, strains and stresses will be covered in order to derive the virtual work principle. The constitutive model for small strain elasto-plasticity will be introduced and implemented.

6.3.1.1 Displacement and strains

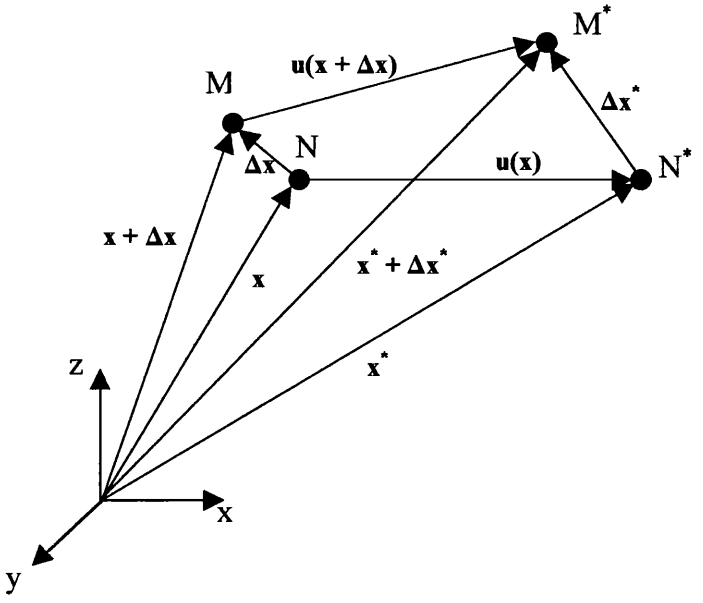


Figure 24 Displacement of the body

The deformation of the body can be seen as a change of the relative distance between two points on the body. After deformation of the body points M and N are moved to M^* and N^*. The distance between these two points has changed from $\Delta \mathbf{x}$ to $\Delta \mathbf{x}^*$. The new distance $\Delta \mathbf{x}^*$ can be (see Figure 24) described in terms of displacements $\mathbf{u}$ as follows

$$\Delta \mathbf{x}^* = \Delta \mathbf{x} + \mathbf{u}(\mathbf{x} + \Delta \mathbf{x}) - \mathbf{u}(\mathbf{x}) \tag{6.36}$$

If $\mathbf{u}$ is differentiable at x then we can perform a Taylor series expansion

$$\mathbf{u}(\mathbf{x} + \Delta \mathbf{x}) = \mathbf{u}(\mathbf{x}) + D\mathbf{u}(\mathbf{x})\Delta \mathbf{x} + O(\Delta \mathbf{x}) \tag{6.37}$$

for sufficiently small $\Delta \mathbf{x}$

$$\lim_{\Delta x \to 0} \frac{\|O(\Delta x)\|}{\|\Delta x\|} = 0 \tag{6.38}$$

so that

$$\mathbf{u}(\mathbf{x} + \Delta \mathbf{x}) - \mathbf{u}(\mathbf{x}) = \nabla \mathbf{u}(\mathbf{x}) \cdot \Delta \mathbf{x} \tag{6.39}$$

Inserting (6.39) into (6.36) and using the infinitessimal form

$$d\mathbf{x}^* = (\mathbf{I} + \nabla \mathbf{u}) d\mathbf{x} \tag{6.40}$$

Square of the length $d\mathbf{x}^*$ leads to

$$d\mathbf{x}^* \cdot d\mathbf{x}^* = (\mathbf{I} + \nabla \mathbf{u}) d\mathbf{x} \cdot (\mathbf{I} + \nabla \mathbf{u}) d\mathbf{x} = d\mathbf{x} (\mathbf{I} + 2\mathbf{E}) d\mathbf{x} \tag{6.41}$$

where $\mathbf{E}$ is the Green-Lagrange strain tensor of the following form

$$\mathbf{E} = \frac{1}{2}\left(\nabla \mathbf{u} + \nabla \mathbf{u}^T + \nabla \mathbf{u}^T \nabla \mathbf{u}\right) \tag{6.42}$$

In the case of a small displacement field the following assumption is valid: if $\|\nabla \mathbf{u}\| << 1$ then $\|\nabla \mathbf{u}^T \nabla \mathbf{u}\| << \|\nabla \mathbf{u}\|$ and hence the product in (6.42) can be neglected. This leads to the infinitessimal strain tensor $\boldsymbol{\varepsilon}$ defined as follows:

$$\boldsymbol{\varepsilon} = \frac{1}{2}(\nabla \mathbf{u} + \nabla \mathbf{u}^T) \tag{6.43}$$

6.3.1.2 Stresses

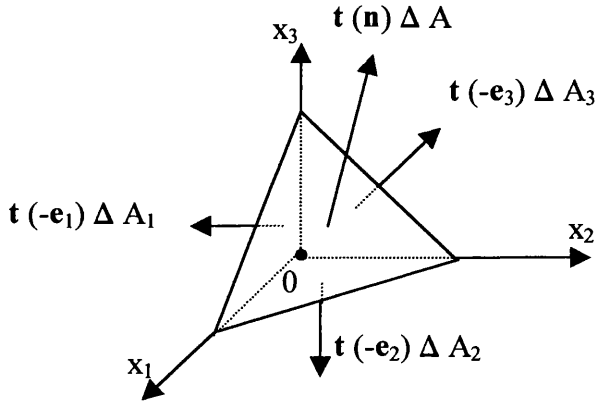


Figure 25 Cauchy tetrahedron

Assume we cut the tetrahedron from the body Ω, which is loaded with body force $\mathbf{f}$. The tetrahedron is limited by the surface A with its normal $\mathbf{n}$ and with A_1, A_2, A_3 whose normals corresponds to unit vectors $(\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3)$. Surface loads are described as

$$\mathbf{t}_j = -\mathbf{t}(\mathbf{x}, -\mathbf{e}_j) \tag{6.44}$$

except on the surface A where the load is $\mathbf{t}_n = \mathbf{t}(\mathbf{x}, \mathbf{n})$.

If we write the local linear momentum balance for the tetrahedron with volume ΔV bounded by surface ΔA:

$$\int_{\Delta V}\mathbf{f}(\mathbf{x})dv+\int_{\Delta A}\mathbf{t}(\mathbf{x},\mathbf{n})da=0 \tag{6.45}$$

assuming sufficiently small volume so that

$$\int_{\Delta V}\mathbf{f}(\mathbf{x})dv\approx\mathbf{f}\Delta V \tag{6.46}$$

$$\int_{\partial A}\mathbf{t}(\mathbf{x},\mathbf{n})da\approx\mathbf{t}(\mathbf{x},\mathbf{n})\Delta A+\sum_{j=1}^{3}\mathbf{t}(\mathbf{x},-\mathbf{e}_j)dA_j=\left[\mathbf{t}(\mathbf{x},\mathbf{n})-\mathbf{t}_j n_j\right]\Delta A \tag{6.47}$$

then

$$\mathbf{t}(\mathbf{x},\mathbf{n})-\mathbf{t}_j n_j+\mathbf{f}\frac{\Delta V}{\Delta A}=0 \tag{6.48}$$

As the tetrahedron shrinks to the point $(\Delta V/\Delta A\rightarrow 0)$ and hence $\mathbf{t}(\mathbf{x},\mathbf{n})=\mathbf{t}_j n_j$ which clearly shows that $\mathbf{t}(\mathbf{x},\bullet)$ is a linear function of $\mathbf{n}$. Defining $\boldsymbol{\sigma}_{ij}:=\mathbf{e}_i\cdot\mathbf{t}_j$ the following relation is obtained

$$\mathbf{t}(\mathbf{x},\mathbf{n})=\boldsymbol{\sigma}(\mathbf{x})\,\mathbf{n} \tag{6.49}$$

where $\boldsymbol{\sigma}$ is the strain tensor. The angular momentum balance for the arbitrary domain H ($H\subset\Omega$) implies symmetry of the stress tensor ($\boldsymbol{\sigma}_{ij}=\boldsymbol{\sigma}_{ji}$).

By applying the divergence theorem to the surface integral term in (6.45)

$$\int_{\partial H}\mathbf{t}(\mathbf{x},\mathbf{n})da=\int_{\partial H}\boldsymbol{\sigma}(\mathbf{x})\mathbf{n}\,da=\int_{H}\nabla\cdot\boldsymbol{\sigma}(\mathbf{x})\,dv \tag{6.50}$$

the linear momentum balance (6.45) can be written as

$$\int_{H}\left[\nabla\cdot\boldsymbol{\sigma}(\mathbf{x})+\mathbf{f}(\mathbf{x})\right]dv=0 \tag{6.51}$$

for all subdomains H bounded by ∂H of domain Ω. Therefore

$$\nabla\cdot\boldsymbol{\sigma}(\mathbf{x})+\mathbf{f}(\mathbf{x})=0 \tag{6.52}$$

for each point $\mathbf{x}$ in Ω. The set of partial differential equations (6.52) represents the local equilibrium equations.

6.3.1.3 Principle of virtual work

In order to obtain a suitable weak form formulation for the finite element method we can multiply the local equilibrium equation by a displacement variation $\delta\mathbf{u}$ and after integration we can obtain

$$\int_{\Omega}[\nabla\cdot\boldsymbol{\sigma}+\mathbf{f}]\delta\mathbf{u}\,d\Omega=0 \tag{6.53}$$

with the natural boundary condition $\boldsymbol{\sigma}\mathbf{n}=\mathbf{t}^s$ on Γ_s and essential boundary condition $\mathbf{u}=0$ on Γ_u.

Using Green's formula on equation (6.53) leads to

$$\int_{\Omega}[-\boldsymbol{\sigma}:\nabla\delta\mathbf{u}+\mathbf{f}\cdot\delta\mathbf{u}]\,d\Omega+\int_{\Gamma_s}\boldsymbol{\sigma}\mathbf{n}\cdot\delta\mathbf{u}\,d\Gamma=0 \tag{6.54}$$

Rearranging (6.54) we obtain the principle of virtual work which represents the basis for all displacement based finite element solutions:

$$\int_{\Omega}\boldsymbol{\sigma}:\delta\boldsymbol{\varepsilon}\,d\Omega=\int_{\Omega}\mathbf{f}\cdot\delta\mathbf{u}\,d\Omega+\int_{\Gamma_s}\boldsymbol{\sigma}\mathbf{n}\cdot\delta\mathbf{u}\,d\Gamma \tag{6.55}$$

6.3.1.4 Constitutive equations

The relation between strains and stresses are described by constitutive equations. The constitutive equation characterizes the material behavior, which may under loading be reversible in the case of elasticity or irreversible in the case of plastic deformations.

6.3.1.4.1 Elasticity

Hook's law represents the elastic behavior of the material

$$\boldsymbol{\sigma}=\mathbf{C}:\boldsymbol{\varepsilon} \tag{6.56}$$

where **C** is a tensor of rank 4. In the case of isotropic linear elasticity Hook's law can be rewritten in terms of Lame's elastic constants as

$$\boldsymbol{\sigma}=\lambda\left(\mathbf{tr}[\boldsymbol{\varepsilon}]\right)\mathbf{I}+2\mu\boldsymbol{\varepsilon} \tag{6.57}$$

6.3.1.4.2 Small strain elasto-plasticity

In the mechanical model used in this work small strain plasticity is considered where the additive split of the total strain tensor into elastic and plastic parts characterizes the deformations of the elasto-plastic material

$$\boldsymbol{\varepsilon}=\boldsymbol{\varepsilon}_e+\boldsymbol{\varepsilon}_p \tag{6.58}$$

The stress σ is governed by the elastic constitutive equation

$$\boldsymbol{\varepsilon}_e = \boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}_p \qquad \boldsymbol{\sigma} = \lambda(\mathbf{tr}[\boldsymbol{\varepsilon}_e])\mathbf{I} + 2\mu\boldsymbol{\varepsilon}_e \tag{6.59}$$

The behavior of a plastic material is described using three properties; the yield condition, flow rule and hardening rule. The hardening rule specifies how the yield function is modified during plastic flow. In the case of ideal plasticity, which will be considered, hardening does not affect the yield condition.
The material yields when the stress satisfies a certain yield condition

$$f(\boldsymbol{\sigma}) = 0 \tag{6.60}$$

According to the value of $f(\boldsymbol{\sigma})$ the state of the material is defined. If $f(\boldsymbol{\sigma}) < 0$ the material is in the elastic state while $f(\boldsymbol{\sigma}) = 0$ represents the plastic state of the material. A response where $f(\boldsymbol{\sigma}) > 0$ is inadmissible.
There are several yield criteria available in the literature but for metals the most common choice is the Von Mises yield condition

$$f(\sigma) = \sqrt{\frac{3}{2}(\mathbf{s} : \mathbf{s})} - \sigma_y \tag{6.61}$$

where $\mathbf{s}$ are the deviatoric stresses ($\mathbf{s} = \boldsymbol{\sigma} - \frac{1}{3}\mathbf{tr}[\boldsymbol{\sigma}]$) and σ_y is the yield stress.
The flow rule relates plastic strain increment to the current stresses and the stress increments. Applying the yield condition to obtain the plastic strain increment is given by the following (associative) flow rule

$$d\boldsymbol{\varepsilon}_p = d\lambda \frac{\partial f}{\partial \boldsymbol{\sigma}} \tag{6.62}$$

where the plastic multiplier $d\lambda$ is a scalar to be determined. The valid plastic response should satisfy the Kuhn-Tucker conditions (loading/unloading) and consistency condition, which are of the following form

$$\begin{aligned} d\lambda &\geq 0 \\ f(\boldsymbol{\sigma}) &\leq 0 \\ d\lambda\, f(\boldsymbol{\sigma}) &\leq 0 \\ d\lambda\, df(\boldsymbol{\sigma}) &\leq 0 \end{aligned} \tag{6.63}$$

6.3.2 Finite element implementation of small strain elasto-plasticity

6.3.2.1 Solution strategy

The unknown of the mechanical problem presented by (6.55) are the displacements **u** which will be discretized in the following manner

$$\mathbf{u}=\mathbf{N}\mathbf{a} \tag{6.64}$$

The solution of the global system of equations will be therefore the vector of unknown displacement parameters **a**. In addition to the unknown displacements the unknown state variables have to be determined according to equation (6.62) which represent an additional system of equations.
Following the general formulation presented in Chapter 4 the described problem can be considered as a coupled transient non-linear system. The problem will be treated by forming two separate sets of equations i.e. global and local.
Let $\mathbf{b}=\{\varepsilon_p,\lambda\}$ be a vector of unknown state variables defined for each integration point, ${}^{p}\mathbf{a}$ a vector of global displacement parameters at the end of the previous time step, ${}^{p}\mathbf{b}$ a vector of state variables at the end of previous time step, $\Psi(\mathbf{a},{}^{p}\mathbf{a},\mathbf{b},{}^{p}\mathbf{b})$ a set of global equations, and $\Phi(\mathbf{a},{}^{p}\mathbf{a},\mathbf{b},{}^{p}\mathbf{b})$ a set of local equations.

The following incremental solution strategy is adopted:

- Calculation of the total strain from displacements
  $\boldsymbol{\varepsilon}=\varepsilon(\mathbf{a})$
- Calculation of the trial elastic strains
  $\boldsymbol{\varepsilon}_e^{trial}=\boldsymbol{\varepsilon}-{}^{p}\boldsymbol{\varepsilon}_p$
- Evaluation of the trial elastic stresses according to Hook's law
  $\boldsymbol{\sigma}(\boldsymbol{\varepsilon}_e^{trial})=\lambda Tr(\boldsymbol{\varepsilon}_e^{trial})\mathbf{I}+\mu\boldsymbol{\varepsilon}_e^{trial}$
- Evaluation of the Von Mises yield condition

$$f(\boldsymbol{\sigma}(\boldsymbol{\varepsilon}_e^{trial}))=\sqrt{\frac{3}{2}\left(\boldsymbol{\sigma}-\frac{1}{3}tr\boldsymbol{\sigma}\right):\left(\boldsymbol{\sigma}-\frac{1}{3}tr\boldsymbol{\sigma}\right)}-\sigma_y$$

- If the material is in the elastic state $f(\boldsymbol{\sigma}(\boldsymbol{\varepsilon}_e^{trial}))<0$ then:

$$f(\sigma(\varepsilon_e^{trial})) < 0 \qquad \begin{cases} \Phi = \begin{Bmatrix} \varepsilon_p - {}^{p}\varepsilon_p \\ d\lambda \end{Bmatrix} = 0 \\ \Psi = \int_{\Omega} \sigma(\varepsilon_e^{trial}) : \dfrac{\partial \varepsilon_e^{trial}}{\partial \mathbf{a}} \, d\Omega + \Psi^{external} = \mathbf{0} \end{cases} \tag{6.65}$$

- For a plastic state $f(\sigma(\varepsilon_e^{trial})) \geq 0$ the local system of equations have to be solved in addition within a local Newton-Raphson iteration loop.

$$f(\sigma(\varepsilon_e^{trial})) \geq 0 \qquad \begin{cases} \Phi = \begin{Bmatrix} \varepsilon_p - {}^{p}\varepsilon_p - d\lambda \dfrac{\partial f}{\partial \sigma} \\ f(\sigma(\varepsilon_e)) \end{Bmatrix} = \mathbf{0} \\ \Psi = \int_{\Omega} \sigma(\varepsilon_e) : \dfrac{\partial \varepsilon_e}{\partial \mathbf{a}} \, d\Omega + \Psi^{external} = \mathbf{0} \end{cases} \tag{6.66}$$

In order to consistently linarize the global system of equations implicit dependencies among global and local variables have to be considered according to

$$\mathbf{K}_{\Phi} \frac{\partial \mathbf{b}}{\partial \mathbf{a}} = -\frac{\partial \Phi}{\partial \mathbf{a}} \tag{6.67}$$

Solving the system for $\partial \mathbf{b}/\partial \mathbf{a}$ implicit dependencies can be obtained for the parameters of $\mathbf{a}$.

6.3.2.2 Symbolic input for small strain elasto-plastic element

The solution strategy presented in 6.3.2.1 was implemented in symbolic input for all mechanical elements. Three and six noded triangle as well as four and eight noded quadrilateral axisymmetic elements were developed with two degrees of freedom per node.
Symbolic input for the four noded small strain elasto-plastic quadrilateral element consists of the following steps:

Step 1: Definition of evolution equations as a function

- The function Φplastic is defined which returns, for a given set of state varibles **b** a system of local evolution equations. The last equation is the yield function $\mathcal{F} = 0$.

```
σεI = {#[1, 1], #[2, 2], #[3, 3], #[1, 2]} &;

SplasticQ[b_] := {

  εp = {{b[1], b[4], 0}, {b[4], b[2], 0}, {0, 0, b[3]}};

  εe = εt - εp;
  σe = Simplify[λ Tr[εe] IdentityMatrix[3] + 2 μ εe];
  σf = SMSFreeze[σe, "IgnoreNumbers"];
  s = σf - 1/3 IdentityMatrix[3] Tr[σf];
  ℱ = √(3/2) √(s s /. List → Plus) - σy;
  𝒩 = Map[If[#1 === 0, 0, SMSD[ℱ, #1]] &, σf, {2}];
  {𝒩, ℱ} = SMSRestore[{𝒩, ℱ}, σf, MapThread[Rule, {σf, σe}, 2] // Flatten];

  Join[σεI[εp - εp0 - b[[5]] 𝒩], {ℱ}]
}
```

Step 2: Initialization

- The AceGen and the Computational templates are initialized and the basic element options are set. The element will have the four noded quadrilateral topology (code Q1) with two global degrees of freedom per integration point (displacement **u** and **v**). The subroutine for the evalua tion of the tangent matrix and residual vector will be generated.

```
SMSInitialize["Mech4", "VectorLength" → 2000, "Mode" -> "Prototype",
  "Language" → "C++"];
SMTInitialize["Mech4", "CDriver", "SMTTopology" → "Q1", "SMTDOFGlobal" → 2,
  "SMTSymmetricTangent" → 0];
SMTUserSubroutine["Tangent and residual"];
```

Step 2: Input data interface

- The coordinates of the element nodes and the current values of the displacements are supplied to the routine. All global degrees of freedom are collected in one single vector **a**.

```
Ri = Array[SMSReal[nd$$[#, "X", 1], {{-1, 1, 1, -1}[[#]] + SMSRandom[]}] &, 4];
Zi = Array[SMSReal[nd$$[#, "X", 2], {{-1, -1, 1, 1}[[#]] + SMSRandom[]}] &, 4];
uti = Array[SMSReal[nd$$[#, "at", 1]] &, 4];
upi = Array[SMSReal[nd$$[#, "ap", 1]] &, 4];
vti = Array[SMSReal[nd$$[#, "at", 2]] &, 4];
vpi = Array[SMSReal[nd$$[#, "ap", 2]] &, 4];
a = Flatten[Transpose[{uti, vti}]];
```

- Material parameters are supplied.

```
SMTGroupDataNames = {"Elastic modulus", "Poisson ratio", "Yield stress"};
{E, ν, σy} = Array[SMSReal[es$$["Data", #]] &, 3];
```

- Start of the loop over integration points where ξ and η are the local coordinates of the current integration point and *wGauss* is its corresponding weight.

```
NoIp = SMSInteger[es$$["id", "NoIntPoints"]];
SMSDo[IpIndex, 1, NoIp];
{ξ, η, wGauss} = Map[SMSReal[es$$["IntPoints", #1, IpIndex]] &, {1, 2, 4}];
```

Step 3: Definiton of the trial functions

- Definition of the shape functions, interpolation of the physical coordinates and global degrees of freedom. Definition of the Jacobian matrix for the isoparametric mapping from global to local coordinates. Gradient of displacement for axial symmetry is defined.

```
Ni = 1/4 { (1 - ξ) (1 - η), (1 + ξ) (1 - η), (1 + ξ) (1 + η), (1 - ξ) (1 + η)};
R = SMSFreeze[Ni.Ri];
Z = SMSFreeze[Ni.Zi];
ϕ = SMSReal[0];
Jm = | SMSD[R, ξ] | SMSD[R, η] |
     | SMSD[Z, ξ] | SMSD[Z, η] | ;
Jd = Det[Jm];
SMSDefineDerivative[{ξ, η}, {R, Z}, SMSInverse[Jm]];
fGauss = 2 π R Jd wGauss ;
ut = Ni.uti; up = Ni.upi;
vt = Ni.vti; vp = Ni.vpi;
coor = {R, Z, ϕ}; transf = {R Cos[ϕ], Z, -R Sin[ϕ]};
GradAxi = SMSDCovariant[ {ut, vt, 0}, transf, coor, {False}];
Dt = (SMSTensorTransformation[GradAxi, transf, coor, {False, True}] /.
    ϕ → 0);
```

Step 4: Definiton of deformations

- Total (εt) and plastic strains (εp0) are defined. Plastic strains are stored in the element history field.

```
εt = 1/2 (Dt + Transpose[Dt]) ;
hindex = SMSInteger[(IpIndex - 1) 5];
bp = SMSReal[Array[ed$$["hp", hindex + #] &, 5]];
εp0 = | bp[1] | bp[4] | 0 |
      | bp[4] | bp[2] | 0 |
      |   0   |   0   | bp[3] | ;
```

- Lame's constants are calculated from the Elastic modulus and Poisson ratio

```
{λ, μ} = SMTHookeToLame[E, ν];
```

Step 4: Local plastic iteration loop

- Here the local iterative loop starts checking the value of the plastic multiplier for the trial value of local state variables **bp** .

```
SMSIf[plastic[bp][[5]] > 10^-8];
```

- Plastic Newton-Raphson loop. The local system of equations is formed and solved for increments of **b** (**Δbt**)until convergence is achieved. The converged solution is stored in **btv**.

```
linear = SMSLogical[False];
bt = bp;
SMSDo[iter, 1, 30, 1, bt];
R = Rplastic[bt];
KR = SMSD[R, bt];
LUDecomp = SMSLUFactor[KR];
Δbt = SMSLUSolve[LUDecomp, -R];
bt = bt + Δbt;
SMSIf[Plus @@ Map[SMSAbs[#] &, Δbt] < 1 / 10^8 || iter == "29"];
SMSVerbatim["Fortran" -> "EXIT", "Mathematica" -> "Break[];",
  "C++" -> "break;"];
SMSEndIf[];
SMSEndDo[];
btv = SMSReal[bt];
SMSExport[btv, Array[ed$$["ht", hindex + #] &, 5]];
```

- The material is not in the plastic state and hence $d\lambda=0$ and **btv** is set to the previous value .

```
SMSElse[];
linear ← SMSLogical[True];
btv ← bp;
SMSExport[ReplacePart[bp, 0, -1], Array[ed$$["ht", hindex + #] &, 5]];
SMSEndIf[btv, LUDecomp, linear];
```

- Local equations are derived for the converged solution.

```
Φ ⊧ Φplastic[btv];
```

Step 5: Global elastic part

- Equations are evaluated for the characteristic node and therefore the loop over the nodes is required

```
SMSDo[i, 1, SMTNoNodes];
```

- Global residual is exported.

```
δϵui = SMSD[ϵt, uti, i];
Φpui ⊧ fGauss ( σe δϵui /. List → Plus);
δϵvi = SMSD[ϵt, vti, i];
Φpvi ⊧ fGauss ( σe δϵvi /. List → Plus);
SMSExport[{Φpui, Φpvi}, Array[p$$[SMTMaxNoDOFNode (i - 1) + #1] &,
   SMTMaxNoDOFNode], "AddIn" → True];
```

- Evaluation of global stiffnes matrix

```
SMSDo[j, 1, SMTNoNodes];
aj ⊧ Map[SMSPart[#, j] &, {uti, vti}];
```

- Global stiffnes matrix in case of elastic response

```
SMSIf[linear];
KΦ ← SMSD[{Φpui, Φpvi}, aj];
```

- Global stiffness matrix in the case of elasto-plastic response.Implicit dependencies ($\partial\mathbf{b}/\partial\mathbf{a}$) are obtained.

```
SMSElse[];
Map[(δbδa ⊧ SMSLUSolve[LUDecomp, -SMSD[Φ, #]];
    SMSDefineDerivative[btv, #, δbδa]) &, aj];
KΦ ← SMSD[{Φpui, Φpvi}, aj];
SMSEndIf[KΦ];
```

- Global stiffnes matrix is exported

```
SMSExport[KΦ,
  Array[s$$[SMTMaxNoDOFNode (i - 1) + #1, SMTMaxNoDOFNode (j - 1) + #2] &,
   {SMTMaxNoDOFNode, SMTMaxNoDOFNode}], "AddIn" → True];
SMSEndDo[];
```

- The loops over the node and integration loop are ended

```
SMSEndDo[];
SMSEndDo[]; (* end of the integration loop*)
```

Step 5: Code generation

```
SMSWrite["Mech4", "Splice" → SMTSplice];
```

6.3.3 Verification of small strain elasto-plastic element

As verification a convergence test was performed instead of comparison with an analytical solution as in the case of thermal and magnetic fields.
In the convergence test the results of a numerical model in a certain point on the structure are monitored while varying the mesh density. The results should converge to the certain value.
The axisymmetric test problem was chosen to perform the convergence test. The geometry is presented on Figure 26. The body is supported in the *x* and *y* direction at the inner surface while the outer surface is loaded with the surface load q in the y direction. In each calculation the displacement **v** was sampled at the point A while the stress component σ_{xx} was sampled at the point B. Structured meshes (see Figure 27) were generated using the *Computational Templates* mesh generator using equal divisions N_{el} in the *x* and *y* directions. On the outer surface of the solid N_{el} surface load elements were placed applying the load $q = 40$ in the *y* direction on the outer surface.

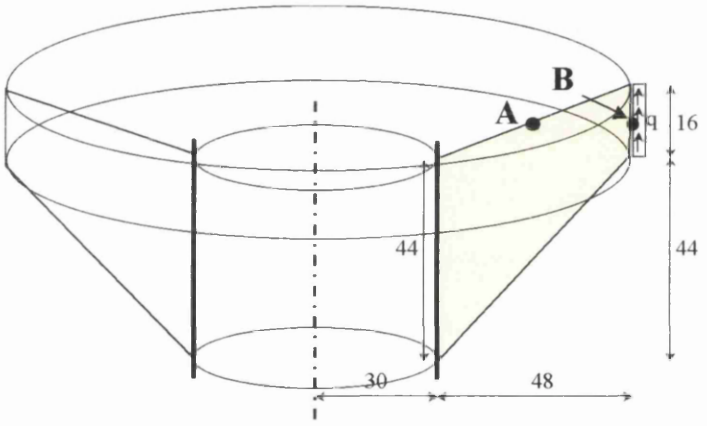


Figure 26 The axisymmetric test problem geometry and loads

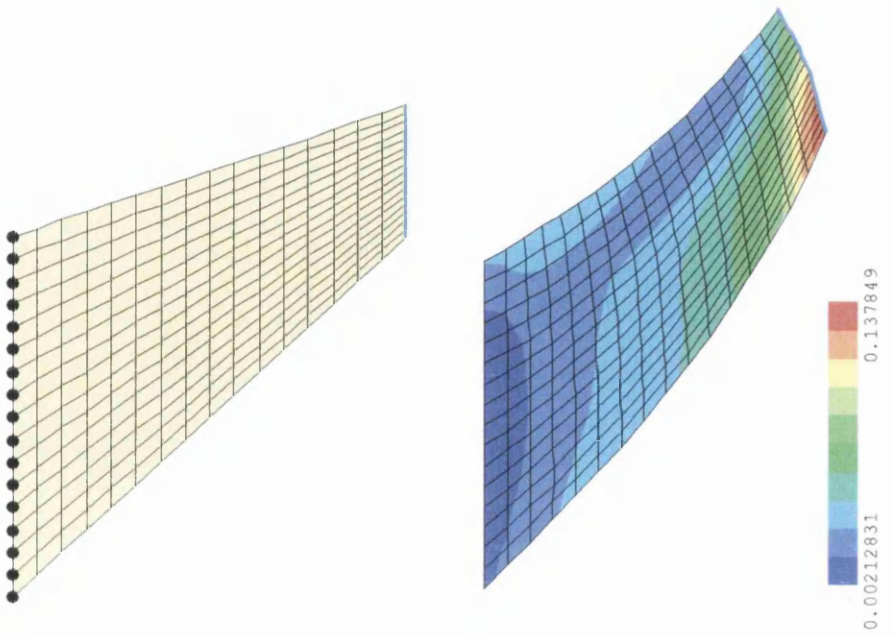


Figure 27 The axisymmetric test problem initial (N_{el}= 16) mesh and results for plastic multiplier λ on the deformed configuration.

In the convergence test the following element topologies were used: three noded triangles (T1), six noded triangles (T2), four noded quadrilaterals (Q1) and eight noded quadrilaterals (Q2).

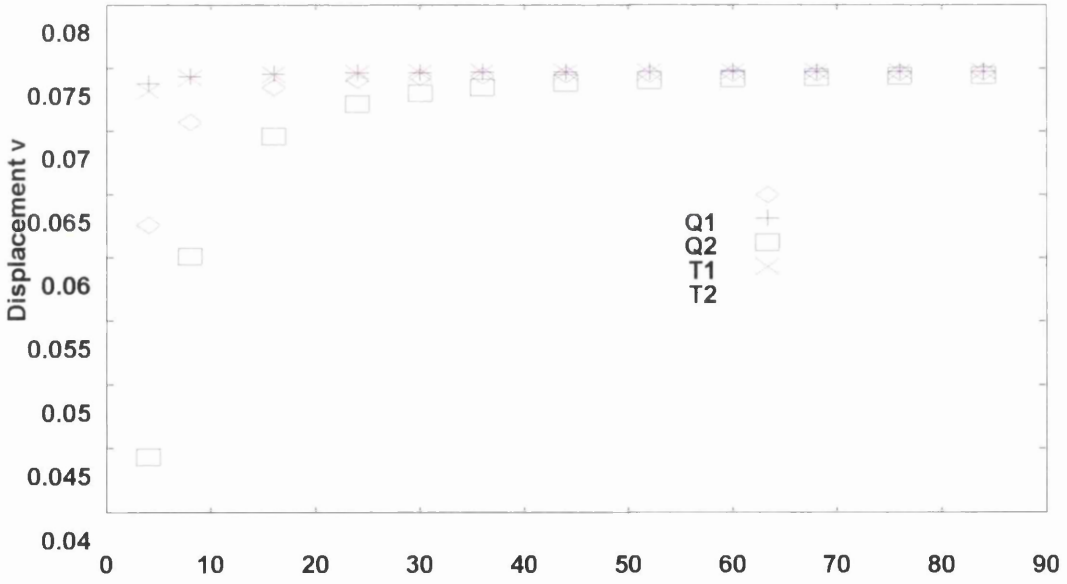


Figure 28 The axisymmetric test problem – Results of convergence test for displacement v at point B.

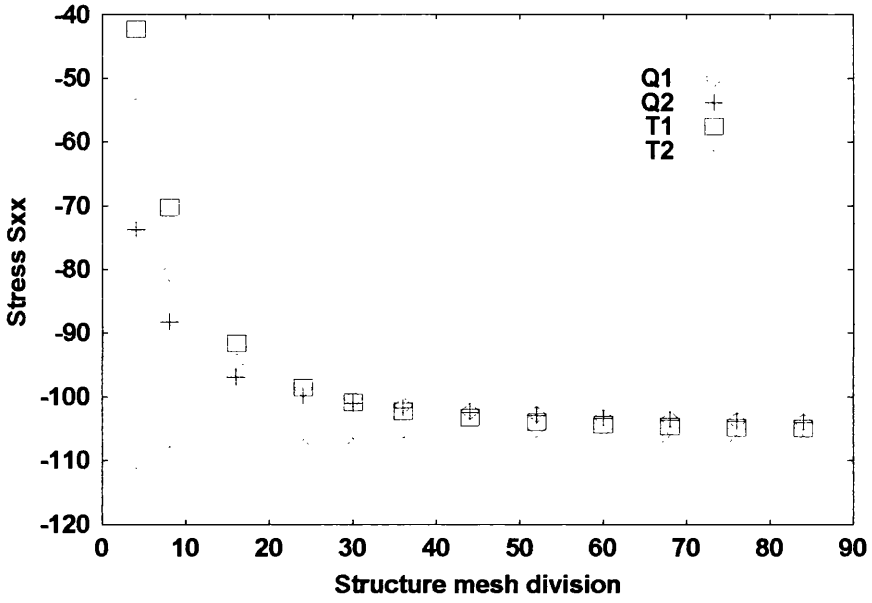


Figure 29 The axisymmetric test problem – Results of convergence test for stress component σ_{xx} at point B.

The results of the convergence test for the displacement at point B are presented in Figure 28 while the stresses σ_{xx} in point A are presented in Figure 29. It is important to note that the results are plotted against structural division, which varied from 4 to 84 and therefore the sizes of meshes were between 16 elements and 7056 elements respectively.

References

[1] J. Bonet, R.D Wood, *Nonlinear Continuum Mechanics for Finite Element Analysis*, Cambridge University Press, Cambridge, 1997.

[2] Y.C. Fung, *Foundation of Solid Mechanics*, Prentice-Hall, 1965

[3] J. Lemaitre, J. L. Chaboche, *Mechanics of solid materials*, Cambridge University Press, 1994.

[4] K.F. Wang, S. Chandrasekar and Henry T.Y. Yang, *Finite-Element Simulation of Induction Heat Treatment*, Journal of Materials Engineering and Performance,Vol.1,97-112,(1992)

[5] C.V. Dodd,W.E. Deeds, *Analytical solution to eddy-current probe coil problems*, Report ORNL-TM-1842, 1967

[6] M.A. Plonus, *Applied Electro-Magnetics*,Mc Graw-Hill,1978

[7] J. Donea,S. Giuliani,A. Philippe, *Finite elements in the solution of electromagnetic induction problems*, Int. Journal for Numerical Methods in Engineering, Vol. 8, 359-367, 1974

[8] C. Chaboudez, S. Clain, R. Glardon, J. Rappaz, M. Swierkosz, R. Touzano, *Numerical modeling of induction heating of long workpieces,* IEEE Transaction on Magnetics, Vol. 30 No. 6, 5026-5036, 1994

[9] G.D. Garbulsky, P. Marino, A. Pignotti, *Numerical model of induction heating of steel-tube ends*, IEEE Transaction on Magnetics, Vol. 33 No. 1, 746-752, 1997

[10] MSC/EMAS ver. 3.0, *User's Manual*, 1994

[11] M.N. Ozisik, *Heat transfer: a basic approach*, McGraw-Hill, 1985

[12] M.N. Ozisik, *Heat conduction*, John Wiley & Sons, 1980

[13] H.S. Carslaw, J.C. Jaeger, *Conduction of heat in solids*, Oxford University Press, 1959

[14] U. Grigull, *Temperaturausgleich in einfachen Körpern*, Springer Verlag, Berlin, Göttingen, Heidelberg, 1964

[15] Hibbitt, Karlsson & Sorensen, Inc., *ABAQUS Theory Manual*, 2000

7 FORMULATION OF MAGNETO-THERMAL-MECHANICAL PROBLEM

In this chapter the formulation of fully coupled problems will be presented. The individual models discussed in the previous chapter will be used to derive the coupled model for inductive heat treatment where the responses of magnetic, thermal and displacement fields are related. Based on the general formulation for coupled problems, presented in Chapter 4, magneto-thermal and magneto-thermal-mechanical elements were derived. Both elements were verified and applied to an example.

7.1 Magneto-Thermal coupling

The electromagnetic field, produced by coil carrying the alternating current, induces eddy currents in the electrically conducting material, which passes trough the coil. As a result of induced eddy currents the material heats resistively. The phenomenon is also known as inductive heating where the thermal response is coupled with the magnetic behavior. The inductive heating is used for a wide variety of practical applications such as heat treatment and surface hardening.

7.1.1 Magneto-Thermal element

The computational model where the magnetic and thermal are fields fully coupled will be presented based on individual models presented in the previous chapter. There are several papers available on this topic[4]-[6].The model is based on the weak form of the equations for the magnetic field:

$$\int_{\Omega}\frac{1}{\mu}\nabla \mathbf{A}\cdot\nabla\delta\mathbf{A}\,d\Omega+\int_{\Omega}\left(\frac{1}{\mu r^{2}}+i\omega\sigma\right)\mathbf{A}\,\delta\mathbf{A}\,d\Omega-\int_{\Omega}\mathbf{J}_{0}\,\delta\mathbf{A}\,d\Omega=0 \qquad (7.1)$$

and thermal field

$$\int_{\Omega} \rho c_p \frac{\partial T}{\partial t} \delta T \, d\Omega + \int_{\Omega} \nabla \delta T \cdot (k \nabla T) d\Omega - \int_{\Omega} q \, \delta T \, d\Omega = 0 \tag{7.2}$$

which will be used to form global residual. Surface integrals were omitted from (7.1) and (7.2). Both fields are related through the heat source q, which is controlled by the magnetic vector potential according to[2]:

$$q = \frac{\sigma \omega^2 \|\mathbf{A}\|^2}{2} \tag{7.3}$$

and temperature dependent material properties.
The system of element equations is formed by grouping (7.1) and (7.2) into one single residual as follows

$$\boldsymbol{\Psi} = \begin{Bmatrix} \boldsymbol{\Psi}_A \\ \boldsymbol{\Psi}_T \end{Bmatrix} = \begin{Bmatrix} \int_{\Omega} \frac{1}{\mu} \nabla A \cdot \nabla \delta A \, d\Omega + \int_{\Omega} \left(\frac{1}{\mu r^2} + i \omega \gamma \right) A \, \delta A \, d\Omega - \int_{\Omega} J_g \delta A \, d\Omega \\ \int_{\Omega} \rho c_p \frac{\partial T}{\partial t} \delta T \, d\Omega + \int_{\Omega} \nabla \delta T \cdot (k \nabla T) d\Omega - \int_{\Omega} q \, \delta T \, d\Omega \end{Bmatrix} = 0 \tag{7.4}$$

The characteristic formulas for submatrix K_{ij} belonging to element node *(i,j)* are defined as follows:

$$\mathbf{K}_{ij} = \begin{bmatrix} \frac{\partial \boldsymbol{\Psi}_{A_i}}{\partial T_j} & \frac{\partial \boldsymbol{\Psi}_{A_i}}{\partial A_j} \\ \frac{\partial \boldsymbol{\Psi}_{T_i}}{\partial T_j} & \frac{\partial \boldsymbol{\Psi}_{T_i}}{\partial A_j} \end{bmatrix} \tag{7.5}$$

One important remark has to be made regarding the residual (7.4) since it contains complex terms. The symbolic system allowed us to solve the problem by automatic separation of the magnetic residual into its imaginary and real part with separate discretization. Therefore the unknown vector potential $A = A_{re} + i A_{im}$ was treated as two separate degrees of freedom A_{re} and A_{im} with its own residuals.
On the basis of (7.4) and (7.5) the axisymmetric magneto-thermal element was generated with three degrees of freedom per node (T, A_{re}, A_{im}). Three noded triangle, four noded quadrilateral and eight noded quadrilateral topologies were implemented. In the implemented elements the following material properties were treated as temperature dependent: electrical conductivity (γ), specific heat (c_p) and thermal conductivity (k). The thermal dependency of each parameter was entered in

tabular form providing data for interpolation of values. The tabular form was replaced with a quadratic interpolation function by the symbolic system.
Symbolic input will be presented only for the steps, which differ from input for the thermal and magnetic element.

```
SMSInitialize["MagThermo4", "VectorLength" → 1000, "Mode" -> "Prototype",
  "Language" → "C++"];
SMTInitialize["MagThermo4", "CDriver", "SMTTopology" → "Q1",
  "SMTDOFGlobal" → 2, "SMTSymmetricTangent" → 0];
SMTUserSubroutine["Tangent and residual"];
```

Step 2: Input data interface

- The current values of the global degrees of freedom are supplied to the routine.

```
ΔT = SMSReal[rdata$$["TimeIncrement"]];
ARei = Array[SMSReal[nd$$[#, "at", 1]] &, SMTNoNodes];
AImi = Array[SMSReal[nd$$[#, "at", 2]] &, SMTNoNodes];
Tti = Array[SMSReal[nd$$[#, "at", 3]] &, SMTNoNodes];
Tpi = Array[SMSReal[nd$$[#, "ap", 3]] &, SMTNoNodes];
```

- Material parameters are supplied.

```
Tem = {20, 100, 200, 300, 400, 500, 600};
γpt = {4000, 3389.830508, 2739.726027, 2207.505519, 1798.561151,
   1483.679525, 1230.0123};
Cpt = {461, 496, 533, 568, 611, 677, 778};
kpt = {41.7, 43.4, 43.2, 41.4, 39.1, 36.7, 34.1};
γtable = Table[{Tem[[i]], γpt[[i]]}, {i, 1, Length[Tem]}];
Cptable = Table[{Tem[[i]], Cpt[[i]]}, {i, 1, Length[Tem]}];
kptable = Table[{Tem[[i]], kpt[[i]]}, {i, 1, Length[Tem]}];
SMTGroupDataNames = {"Permeability μ", "Frequency ω", "Source current J",
   "Density ρ", "Source q"};
{μmg, ωeg, Jgg} = Array[SMSReal[es$$["Data", #]] &, 3];
{ρg, qhg} = Array[SMSReal[es$$["Data", 3 + #]] &, 2];
```

- Start of the loop over integration points where ξ and η are the local coordinates of the current integration point and *wGauss* is its corresponding weight.

```
NoIp = SMSInteger[es$$["id", "NoIntPoints"]];
SMSDo[IpIndex, 1, NoIp];
{ξ, η, fGauss} = Map[SMSReal[es$$["IntPoints", #1, IpIndex]] &, {1, 2, 4}];
```

Step 3: Definiton of the trial functions

- Interpolation of material data

```
γeg = Fit[SetPrecision[γtable, SMSEvaluatePrecision], {1, Tt, Tt^2}, Tt];
Chg = Fit[SetPrecision[Cptable, SMSEvaluatePrecision], {1, Tt, Tt^2}, Tt];
khg = Fit[SetPrecision[kptable, SMSEvaluatePrecision], {1, Tt, Tt^2}, Tt];
```

Step 4: Magneto-Thermal quations

- The aagnetic equation is evaluated for the characteristic node and therefore the loop over the nodes is required

```
SMSDo[i, 1, SMTNoNodes];
```

- Magnetic and thermal equations

```
A = ARe + I AIm;
δA = δARe + I δAIm;
ΦA =
  fGauss ( 1/μmg ((SMSD[A, {R, Z}].SMSD[δA, {R, Z}])) + (1/(μmg R^2) + I ωeg γeg) A δA -
    Jgg δA);
ΦAi = ComplexExpand[ΦA];
ΦARei = ΦAi /. I → 0;
ΦAImi = ΦAi - ΦARei /. I → 1;
Δqmagn = γeg ωeg^2 (ARe^2 + AIm^2) / 2;
δTi = SMSD[Tt, Tti, i];
ΦTi =
  fGauss (ρg Chg (Tt - Tp) / ΔT δTi + khg SMSD[δTi, {R, Z}].SMSD[Tt, {R, Z}] -
    (qhg + Δqmagn) δTi);
```

- The element residual vector is exported as a routine output.

```
SMSExport[{ΦARei, ΦAImi, ΦTi},
  Array[p$$[SMTMaxNoDOFNode (i - 1) + #1] &, SMTMaxNoDOFNode], "AddIn" → True];
```

- The element sfiffness matrix is exported as a routine output.

```
SMSDo[j, 1, SMTNoNodes];
ARej = SMSPart[ARei, j];
AImj = SMSPart[AImi, j];
Ttj = SMSPart[Tti, j];
KΦ = SMSD[{ΦARei, ΦAImi, ΦTi}, {ARej, AImj, Ttj}];
SMSExport[KΦ,
  Array[s$$[SMTMaxNoDOFNode (i - 1) + #1, SMTMaxNoDOFNode (j - 1) + #2] &,
   {SMTMaxNoDOFNode, SMTMaxNoDOFNode}], "AddIn" → True];
SMSEndDo[];
```

- The loops over the node and integration loop are ended

```
SMSEndDo[];
SMSEndDo[]; (* end of the integration loop*)
```

Step 5: Code generation

```
SMSWrite["MagThermo4", "Splice" → SMTSplice];
```

7.1.2 Verification of magneto-thermal element

The verification of the magneto-thermal element was performed using the example of inductive heating of a half space, which was already used for verification of the magnetic element (see Section 5.2.3). The solution of the temperature field for a half space geometry with internal heat source described by a function $q(r,z)$ is available in the literature[2][3]. Temperature at a point with coordinates (r,z) at time t is evaluated according to the following equation:

$$T(r,z,t) = T_0 + \frac{2\alpha}{\pi k}\int_{\tau=0}^{t}\int_{r'=0}^{\infty}\int_{z'=0}^{\infty}\int_{\beta=0}^{\infty}\int_{\eta=0}^{\infty}\beta r' J_0(\beta r)\cos(\mu z)\cos(\mu z')e^{-\alpha(\beta^2+\mu^2)(t-\tau)} \\ q(r',-z')d\beta\, d\eta\, dr'\, dz'\, d\tau \tag{7.6}$$

with the following boundary conditions ($\partial T/\partial r=0$ at $r=0$ and $\partial T/\partial z=0$ at $z=0$). The initial temperature is denoted T_0, $\alpha=k/\rho c_p$ and J_0 is a Bessel function.

Due to the high computation time needed for complete evaluation of the analytical solution the results for the magnetic vector potential were interpolated over the domain and the magnetic heat source was then calculated according to (7.3) and introduced into (7.6) as a volume heat source *(q(r,z))*. Graphical representation of the absolute value of the magnetic vector potential in a half-space is presented in Figure 30.

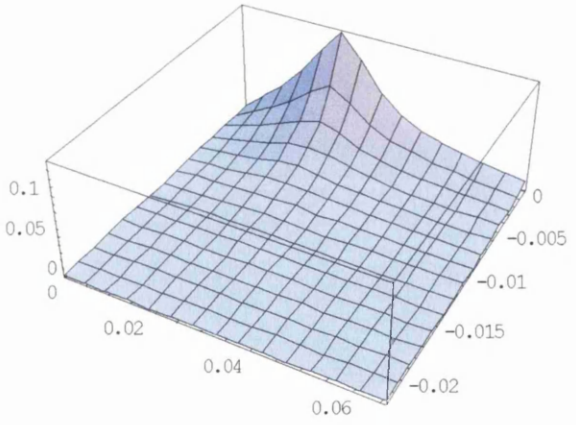


Figure 30 Distribution of absolute value of the magnetic vector potential in a half-space

The material parameters used in the verification procedure are presented in Table 18.

Parameter	Halfspace	Coil	Air
μ	$90\ \mu_0$	μ_0	μ_0
ω	60	60	60
γ	$3\ 10^6$	$5.7\ 10^7$	0
J_g	0	$1.2\ 10^{10}$	0
ρ	7500	8933	1.1614
c_p	650	385	1000
k	35	390	0.0243
q	0	0	0

Table 18 Material parameters used in verification of the magneto-thermal element

The element used for the verification process was slightly modified since the material data were constant and therefore they were all passed to the routine as group data. Numerical results were in good agreement with the analytical solution as presented in Figure 31.

The influence of temperature dependent material properties was also analyzed. Using the material data for Steel CK- 45 once at room temperature and once as temperature dependent the results presented in Figure 32 were obtained. It is clear, from the presented results, that the heating process is slower in the case where the material properties were treated as temperature dependent. A slower heating rate is a result of the electric conductivity influence since its value tends to fall significantly with higher temperature.

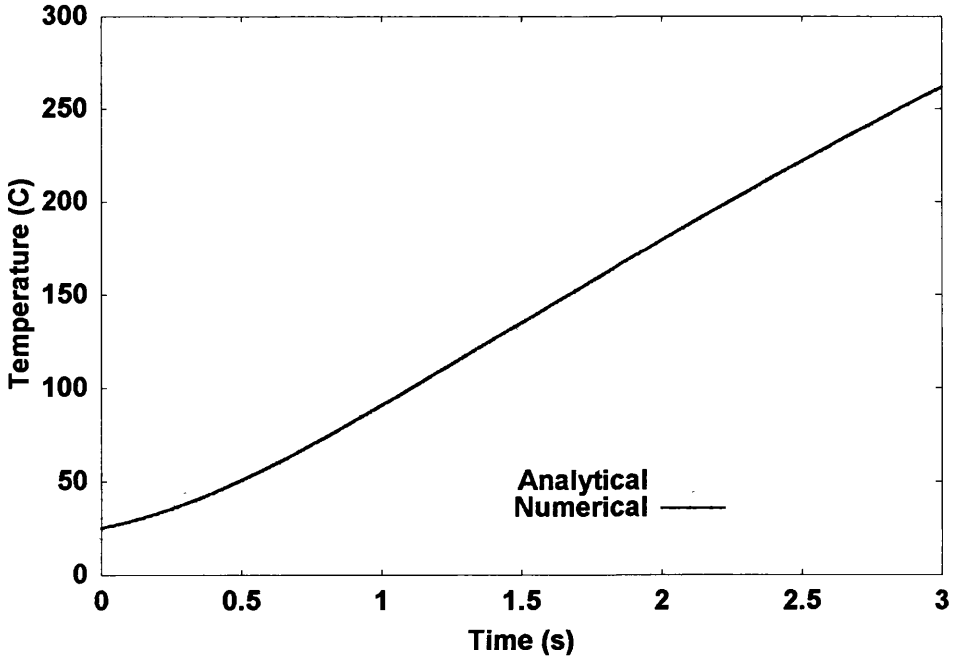


Figure 31 Comparison of numerical solution against analytical solution for point T(0.024,0.006)

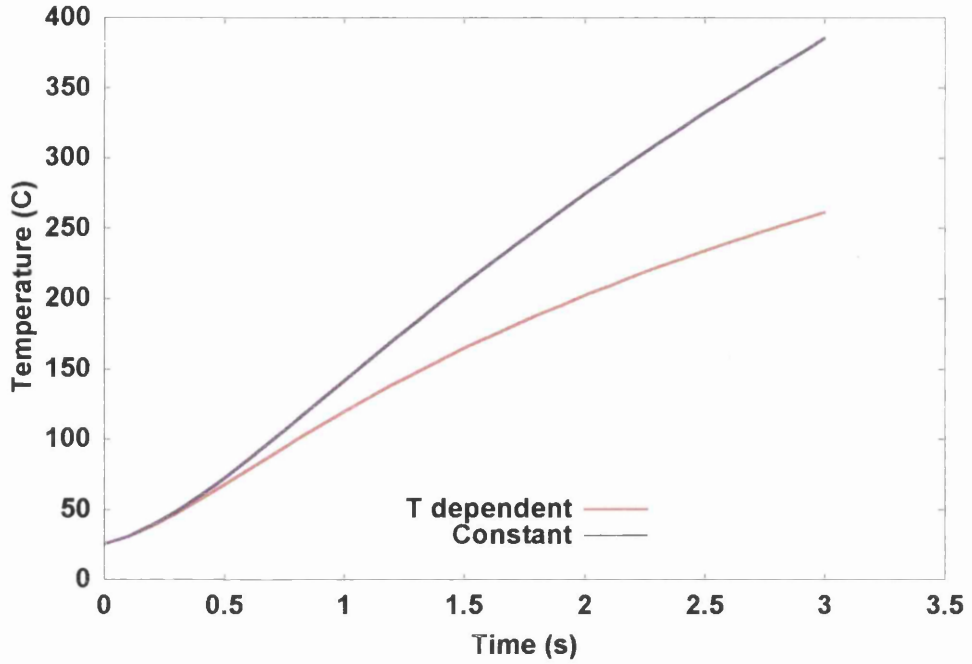


Figure 32 Comparison of numerical results including and excluding thermal dependency of material parameters

The convergence diagrams for four and eight noded quadrilateral element were also created. The results are in agreement with theoretical expectations as presented in Figure 33 where the percentage error is plotted against the number of degrees of freedom on the logarithmic scale.

The error was calculated according to the analytical solution for temperature at point *T(0.024,-0.006)*. Some of the meshes used in the convergence diagram are presented in Figure 34.

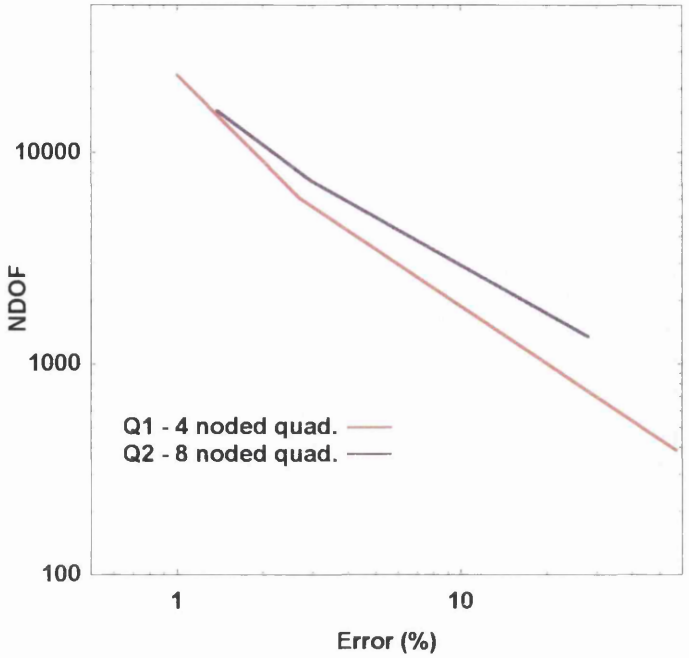


Figure 33 Convergence diagram for magneto-thermal 4 and 8 noded quadrilateral

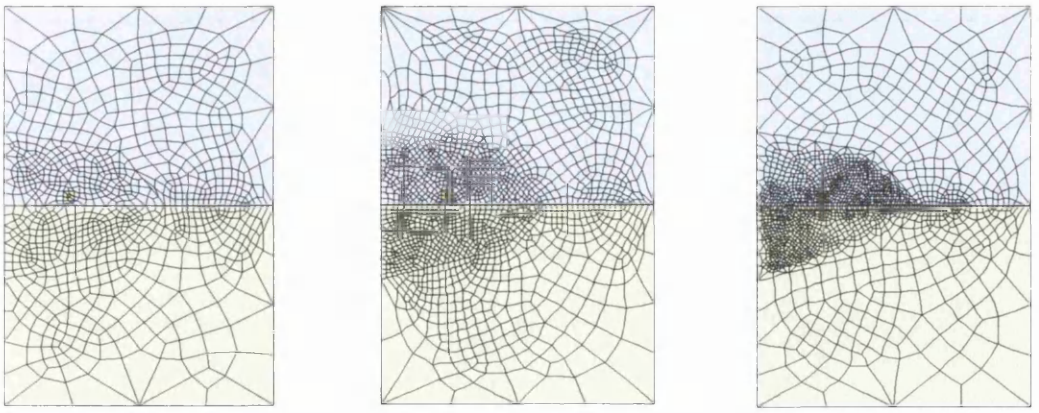


Figure 34 Convergence diagram – different meshes

7.2 Magneto-Thermal-Mechanical coupling

As a consequence of the inductive heating process thermal strains and corresponding stresses are introduced into the material. The proper distribution of stresses after inductive heating is crucial for success of the heat treatment process. In order to obtain the stress distribution in the workpiece during the heat treatment process the computational model couples magnetic, thermal and displacement field effects.

7.2.1 Magneto-thermo-mechanical element

The magneto-thermal element presented in the previous section has to be modified to include the small strain elasto-plastic formulation presented in the previous chapter (see Section 6.4.2.1). In this case the residual should include three coupled fields, with the following set of unknowns for node *i*:

$$\mathbf{a}=(A_{im},A_{re},T,u,v) \tag{7.7}$$

and additional set of unknowns at the integration point *k*:

$$\mathbf{b}=(\varepsilon_{xx}^{pl},\varepsilon_{yy}^{pl},\varepsilon_{zz}^{pl},\varepsilon_{xy}^{pl},\lambda) \tag{7.8}$$

The developed elasto-plastic model should be modified to include additional thermal strains as follows:

$$\boldsymbol{\varepsilon}_t=\alpha(T-T_o)\mathbf{I} \tag{7.9}$$

and hence $\boldsymbol{\varepsilon}=\frac{1}{2}(\nabla\mathbf{u}+\nabla\mathbf{u}^T)+\boldsymbol{\varepsilon}_t$ where α is a thermal coefficient.

The complete system of element equations is of the following form

$$f(\boldsymbol{\sigma}(\boldsymbol{\varepsilon}_e^{trial}))\geq 0 \begin{cases} \boldsymbol{\Phi}=\begin{Bmatrix}\boldsymbol{\varepsilon}_p-{}^{p}\boldsymbol{\varepsilon}_p-\lambda\dfrac{\partial f}{\partial\boldsymbol{\sigma}}\\ f(\boldsymbol{\sigma}(\boldsymbol{\varepsilon}_e^{trial}))\end{Bmatrix}=0 \\ \boldsymbol{\Psi}=\begin{Bmatrix}\int_\Omega \rho c_p\dfrac{\partial T}{\partial t}\delta T\,d\Omega+\int_\Omega\nabla\delta T\cdot(k\nabla T)d\Omega-\int_\Omega q\,\delta T\,d\Omega\\ \int_\Omega\dfrac{1}{\mu}\nabla A\cdot\nabla\delta A\,d\Omega+\int_\Omega(\dfrac{1}{\mu r^2}+i\omega\gamma)A\,\delta A\,d\Omega-\int_\Omega J_g\delta A\,d\Omega\\ \int_\Omega\boldsymbol{\sigma}(\boldsymbol{\varepsilon}_e^{trial}):\dfrac{\partial\boldsymbol{\varepsilon}_e^{trial}}{\partial\mathbf{u}}\,d\Omega\end{Bmatrix}+\boldsymbol{\Psi}^{external}=0 \end{cases} \tag{7.10}$$

$$f(\sigma(\varepsilon_e^{trial}))<0 \qquad \begin{cases} \Phi = \begin{Bmatrix} \varepsilon_p - {}^{p}\varepsilon_p \\ d\lambda \end{Bmatrix} = 0 \\ \Psi = \begin{Bmatrix} \int_\Omega \rho c_p \dfrac{\partial T}{\partial t}\delta T\, d\Omega + \int_\Omega \nabla \delta T \cdot (k\nabla T) d\Omega - \int_\Omega q\,\delta T\, d\Omega \\ \int_\Omega \dfrac{1}{\mu}\nabla A \cdot \nabla \delta A\, d\Omega + \int_\Omega \left(\dfrac{}{\mu r^2} + i\omega\gamma\right) A\,\delta A\, d\Omega - \int_\Omega J_g \delta A\, d\Omega \\ \int_\Omega \sigma(\varepsilon_e^{trial}) : \dfrac{\partial \varepsilon_e^{trial}}{\partial \mathbf{u}}\, d\Omega \end{Bmatrix} + \Psi^{external} = 0 \end{cases} \tag{7.11}$$

The symbolic input for the magneto-thermal-mechanical element is provided below.

Step 1: Definition of evolution equations as a function

- The function Φplastic is defined which returns, for a given set of state varibles **b** a system of local evolution equations. The last equation is the yield function $\mathcal{F} = 0$.

```
σεI = {#[1, 1], #[2, 2], #[3, 3], #[1, 2]} &;

σεII = | #[1] | #[4] | 0 |
       | #[4] | #[2] | 0 |  &;
       |  0   |  0   | #[3] |

Φplastic[b_] := (

  εp = ( | b[1] | b[4] | 0 | ) ;
         | b[4] | b[2] | 0 |
         |  0   |  0   | b[3] |

  εe = εt - εp;
  σe = Simplify[λ Tr[εe] IdentityMatrix[3] + 2 μ εe];
  σf = SMSFreeze[σe, "IgnoreNumbers"];
  s = σf - 1/3 IdentityMatrix[3] Tr[σf];
  F = √(3/2) √(s s /. List → Plus) - σy;
  A = Map[If[#1 === 0, 0, SMSD[F, #1]] &, σf, {2}];
  {A, F} = SMSRestore[{A, F}, σf, MapThread[Rule, {σf, σe}, 2] // Flatten];

  Join[σεI[εp - εp0 - b[[5]] A], {F}]
)
```

Step 2: Initialization

- The AceGen and the Compuational templates are initialized and the basic element options are set. The element will have the four noded quadrilateral topology (code Q1) with five global degrees of freedom per node (real and imaginary component of the magnetic vector potential (**Are,Aim**), temperature **T** and displacements (**u** , **v**). The subrutine for the evaluation of the tangent matrix and residual vector will be generated.

```
SMSInitialize["MagThermoMech4", "VectorLength" → 2000, "Mode" -> "Optimal",
  "Language" → "C++"];
SMTInitialize["MagThermoMech4", "CDriver", "SMTTopology" → "Q1",
  "SMTDOFGlobal" → 5, "SMTSymmetricTangent" → 0,
  "SMTNoTimeStorage" → 6 es$$["id", "NoIntPoints"]];
SMSSearchPrecision = 60; SMSEvaluatePrecision = 80;
SMTUserSubroutine["Tangent and residual"];
```

- List of temperature dependent material parameters are supplied.

```
Tem = {20, 100, 200, 300, 400, 500, 600, 1200};
γpt = {4000, 3389.830508, 2739.726027, 2207.505519, 1798.561151,
   1483.679525, 1230.0123, 900.};
Cpt = {461, 496, 533, 568, 611, 677, 778, 850};
kpt = {41.7, 43.4, 43.2, 41.4, 39.1, 36.7, 34.1, 25.};
Efit = {213., 212., 207., 199., 192., 184., 175., 164., 80.};
Syfit = {610., 600., 500., 400., 380., 350., 320., 300., 80.};
γtable = Table[{Tem[[i]], γpt[[i]]}, {i, 1, Length[Tem]}];
Cptable = Table[{Tem[[i]], Cpt[[i]]}, {i, 1, Length[Tem]}];
kptable = Table[{Tem[[i]], kpt[[i]]}, {i, 1, Length[Tem]}];
Etable = Table[{Tem[[i]], Efit[[i]]}, {i, 1, Length[Tem]}];
Sytable = Table[{Tem[[i]], Syfit[[i]]}, {i, 1, Length[Tem]}];
```

Step 2: Input data interface

- The coordinates of the element nodes and the current values of the magnetic vector potential, temperature and displacements are supplied to the routine. All global degrees of freedom are collected in one single vector **a**.

```
Ri = Array[ SMSReal[nd$$[#, "X", 1], {{-1, 1, 1, -1}[[#]] + SMSRandom[]}] &,
   SMTNoNodes];
Zi = Array[ SMSReal[nd$$[#, "X", 2], {{-1, -1, 1, 1,}[[#]] + SMSRandom[]}] &,
   SMTNoNodes];
ΔT = SMSReal[rdata$$["TimeIncrement"]];
ARei = Array[SMSReal[nd$$[#, "at", 1]] &, SMTNoNodes];
AImi = Array[SMSReal[nd$$[#, "at", 2]] &, SMTNoNodes];
Tti = Array[SMSReal[nd$$[#, "at", 3]] &, SMTNoNodes];
Tpi = Array[SMSReal[nd$$[#, "ap", 3]] &, SMTNoNodes];
uti = Array[SMSReal[nd$$[#, "at", 4]] &, SMTNoNodes];
upi = Array[SMSReal[nd$$[#, "ap", 4]] &, SMTNoNodes];
vti = Array[SMSReal[nd$$[#, "at", 5]] &, SMTNoNodes];
vpi = Array[SMSReal[nd$$[#, "ap", 5]] &, SMTNoNodes];
a = Flatten[Transpose[{ARei, AImi, Tti, uti, vti}]];
```

- Material parameters are supplied.

```
SMTGroupDataNames = {"Permeability", "Frequency omega", "Source current J",
   "Density Ro", "Source q", "C1", "C2", "Tcurie", "Poisson ration",
   "Thermal koef. alfa", "Reference temp. T0"};
{μ0, ωeg, Jgg, ρg, qhg, c1, c2, Tcurie, ν, αg, T0} =
  Array[SMSReal[es$$["Data", #]] &, 11];
```

- Start of the loop over integration points where ξ and η are the reference coordinates of the current integration point and *wGauss* is its corresponding weight.

```
NoIp = SMSInteger[es$$["id", "NoIntPoints"]];
SMSDo[IpIndex, 1, NoIp];
{ξ, η, wGauss} = Map[SMSReal[es$$["IntPoints", #1, IpIndex]] &, {1, 2, 4}];
```

Step 3: Definiton of the trial functions

- Definition of the shape functions, interpolation of the physical coordinates and global degrees of freedom. Definition of Jacobian matrix for the isoparametric mapping from global to local coordinates.

```
Ni = 1/4 {(1 - ξ) (1 - η), (1 + ξ) (1 - η), (1 + ξ) (1 + η), (1 - ξ) (1 + η)};
R = SMSFreeze[Ni.Ri];
Z = SMSFreeze[Ni.Zi];
ϕ = SMSReal[0];
Jm = | SMSD[R, ξ]  SMSD[R, η] |
     | SMSD[Z, ξ]  SMSD[Z, η] |;
Jd = Det[Jm];
SMSDefineDerivative[{ξ, η}, {R, Z}, SMSInverse[Jm]];
fGauss = 2 π R Jd wGauss ;
ARe = Ni.ARei; AIm = Ni.AImi;
Tt = Ni.Tti; Tp = Ni.Tpi;
ut = Ni.uti; up = Ni.upi;
vt = Ni.vti;
vp = Ni.vpi;
```

- Interpolation functions for the temperature dependent material parameters.

```
γeg = 1000 Fit[SetPrecision[γtable, SMSEvaluatePrecision],
    {1, Tt, Tt^2, Tt^3}, Tt];
Chg = Fit[SetPrecision[Cptable, SMSEvaluatePrecision], {1, Tt, Tt^2, Tt^3}, Tt];
khg = Fit[SetPrecision[kptable, SMSEvaluatePrecision], {1, Tt, Tt^2, Tt^3}, Tt];
μmg = -c1 / π * μ0 * ArcTan[c2 (Tt - Tcurie)] + c1 / 2 * μ0 + μ0;
E = 1000000000 Fit[SetPrecision[Etable, SMSEvaluatePrecision],
    {1, Tt, Tt^2, Tt^3}, Tt];
σy = 1000000 Fit[SetPrecision[Sytable, SMSEvaluatePrecision],
    {1, Tt, Tt^2, Tt^3}, Tt];
```

Step 4: Definiton of deformations

- Total (ϵt) and plastic strains (ϵp0) are defined. Plastic strains are stored in the element history field.

```
coor = {R, Z, ϕ}; transf = {R Cos[ϕ], Z, -R Sin[ϕ]};
GradAxi = SMSDCovariant[{ut, vt, 0}, transf, coor, {False}];
Dt = (SMSTensorTransformation[GradAxi, transf, coor, {False, True}] /.
    ϕ → 0);
ϵt = 1/2 (Dt + Transpose[Dt]) - αg (Tt - T0) IdentityMatrix[3];
hindex = SMSInteger[(IpIndex - 1) 6];
bp = SMSReal[Array[ed$$["hp", hindex + #] &, 5]];
ϵp0 = ( | bp[1]  bp[4]  0 |
        | bp[4]  bp[2]  0 |
        |   0       0   bp[3] | );
```

- Lame's constants are calculated from the Elastic modulus and Poisson ratio

```
{λ, μ} = SMTHookeToLame[E, ν];
```

Step 4: Local plastic iteration loop

- Here the local iterative loop starts checking the value of the plastic multiplier for the trial value of local state variables **bp** .

```
SMSIf[plastic[bp][[5]] > 10^-8];
```

- loop initialisation (dλ=0 and plastic strains are set to the previous value)

```
linear = SMSLogical[True];
btv = ReplacePart[bp, 0, -1];
SMSExport[btv, Array[ed$$["ht", hindex + #] &, 5]];
linear = SMSLogical[False];
bt = ReplacePart[bp, 0, -1];
```

- Plastic Newton-Raphson iteration loop

```
SMSDo[iter, 1, 30, 1, bt];
Ψ := Ψplastic[bt];
KΨ := SMSD[Ψ, bt];
LUDecomp = SMSLUFactor[KΨ];
Δbt := SMSLUSolve[LUDecomp, -Ψ];
bt = bt + Δbt;
SMSIf[Sqrt[Δbt.Δbt] < 1/10^8 || iter == "29"];
    SMSIf[iter == "29"];
        SMSExport[{2, SMSInteger[idata$$["SubDivergence"]] + 1},
        {idata$$["ErrorStatus"], idata$$["SubDivergence"]}];
    SMSEndIf[];
        SMSVerbatim["C++" -> "break;"];
SMSEndIf[];
SMSEndDo[];
```

- The converged solution is stored in **btv**.

```
btv = SMSReal[bt];
SMSExport[btv, Array[ed$$["ht", hindex + #] &, 5]];
```

- The material is not in the plastic state and hence dλ=0 and **btv** is set to the previous value .

```
SMSElse[];
linear = SMSLogical[True];
btv = ReplacePart[bp, 0, -1];
SMSExport[btv, Array[ed$$["ht", hindex + #] &, 5]];
```

- end of local iteration loop

```
SMSEndIf[btv, LUDecomp, linear];
```

- Local equations are derived for the converged solution.

```
Ψ := Ψplastic[btv];
```

Step 5: Global iteration loop

- Equations are evaluated for a characteristic node and therefore the loop over the nodes is required

```
SMSDo[i, 1, SMTNoNodes];
```

- Residual of magnetic part

```
δARe := SMSD[ARe, ARei, i];
δAIm := SMSD[AIm, AImi, i];
A = ARe + I AIm;
δA = δARe + I δAIm;
ωcmp := 2 π ωeg ;
ΨA =
  fGauss ( 1/(μmg) ((SMSD[A, {R, Z}].SMSD[δA, {R, Z}])) +
      ( 1/(μmg R^2) + I ωcmp γeg ) A δA - Jgg δA );
ΨAi = ComplexExpand[ΨA];
ΨARei := (ΨAi /. I → 0);
ΨAImi := (ΨAi - (ΨAi /. I → 0) /. I → 1);
```

- Residual of thermal part

```
Δqmagn = γeg ωcmp^2 (ARe^2 + AIm^2) / 2 ;
δTi = SMSD[Tt, Tti, i];
ΨTi =
  fGauss (ρg Chg (Tt - Tp) / ΔT δTi + khg SMSD[δTi, {R, Z}].SMSD[Tt, {R, Z}] -
      (qhg + Δqmagn) δTi);
```

- Residual of mechanical part

```
δεui = SMSD[εt, uti, i];
Ψpui = fGauss ( σe δεui /. List → Plus);
δεvi = SMSD[εt, vti, i];
Ψpvi = fGauss ( σe δεvi /. List → Plus);
```

- Formation and export of the global residual

```
SMSExport[{ΨARei, ΨAImi, ΨTi, Ψpui, Ψpvi},
  Array[p$$[SMTMaxNoDOFNode (i - 1) + #1] &, SMTMaxNoDOFNode], "AddIn" → True];
```

- Evaluation of the global stiffnes matrix

```
SMSDo[j, 1, SMTNoNodes];
aj = Map[SMSPart[#, j] &, {ARei, AImi, Tti, uti, vti}];
```

- The global stiffnes matrix in the case of elastic response

```
SMSIf[linear];
KΨ = SMSD[{ΨARei, ΨAImi, ΨTi, Ψpui, Ψpvi}, aj];
```

- The global stiffnes matrix in the case of elasto-plastic response.Implicit dependencies ($\partial \mathbf{b}/\partial \mathbf{a}$) are obtained.

```
SMSElse[];
Map[(δbδa = SMSLUSolve[LUDecomp, -SMSD[Ψ, #]];
    SMSDefineDerivative[btv, #, δbδa]) &, aj];
KΨ = SMSD[{ΨARei, ΨAImi, ΨTi, Ψpui, Ψpvi}, aj];
SMSEndIf[KΨ];
```

- Global stiffness matrix is exported

```
SMSExport[KΨ,
  Array[s$$[SMTMaxNoDOFNode (i - 1) + #1, SMTMaxNoDOFNode (j - 1) + #2] &,
    {SMTMaxNoDOFNode, SMTMaxNoDOFNode}], "AddIn" → True];
SMSEndDo[];
```

- The loops over the node and integration loop are ended

```
SMSEndDo[];
SMSEndDo[]; (* end of the integration loop*)
```

Step 5: Code generation

```
SMSWrite["Splice" → SMTSplice];
```

7.2.2 Verification of Magneto-Thermal-Mechanical element

Due to the complexity of the model the verification was performed using a convergence test since experimental data for such problems are not easy to obtain. The convergence tests were performed on the example illustrated in Figure 35 where a cylindrical rod is heated by a single coil. Tests were performed using a structured mesh for both quadrilateral and triangular elements as presented in Figure 36.

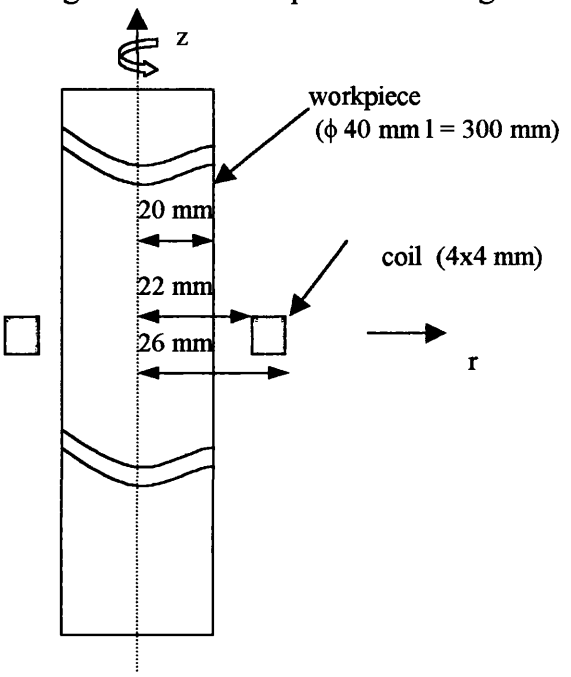


Figure 35 Schematic of the example used in the verification procedure

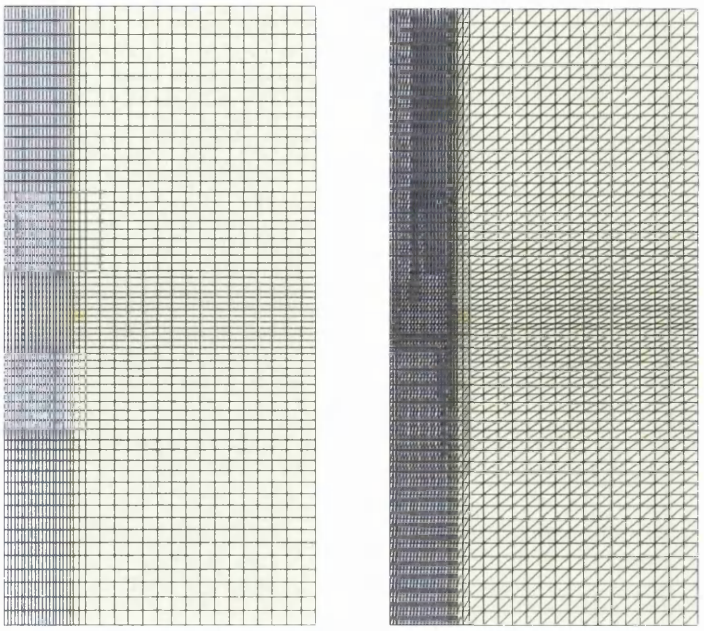


Figure 36 Examples of finite element meshes using in the verification procedure

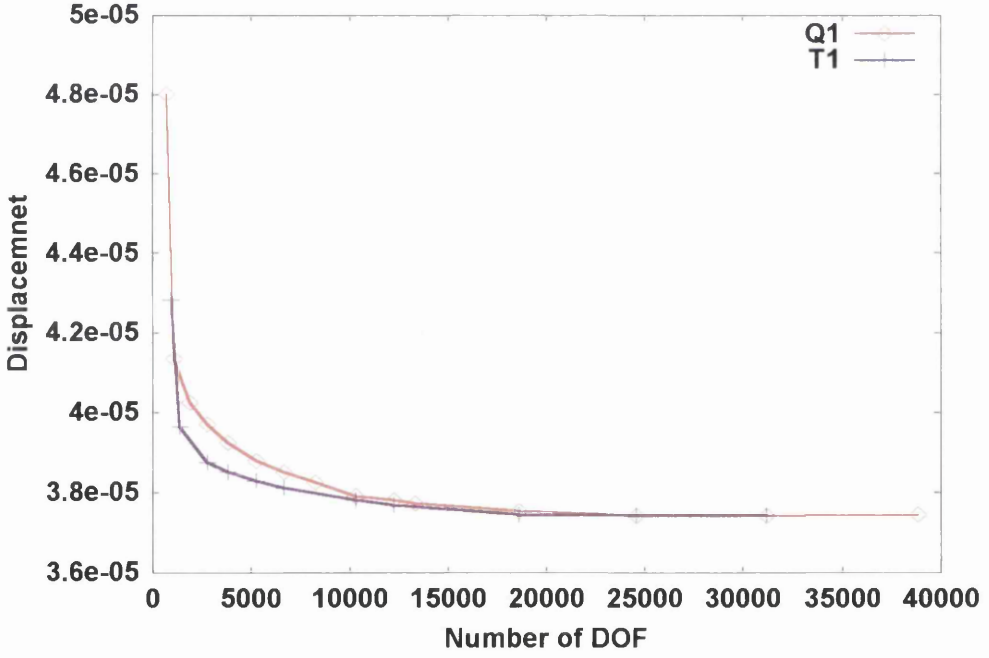


Figure 37 Results of convergence test for magneto-thermal-mechanical element – convergence of displacement in the radial direction at point *(0.01,0.02)*.

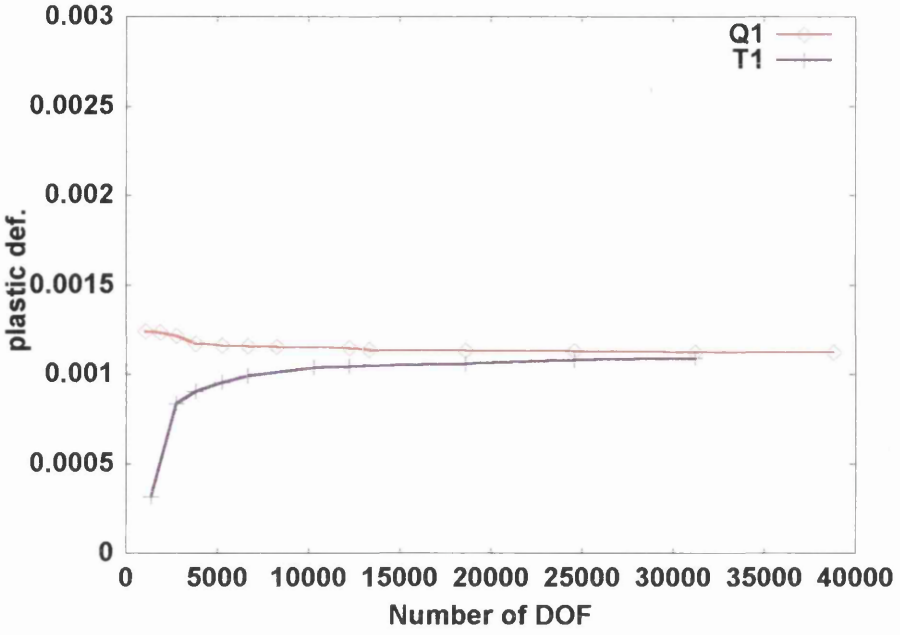


Figure 38 Results of convergence test for magneto-thermal-mechanical element – convergence of the plastic deformation ε_{xx}^{pl} at point *(0.01,0.0)*.

The results of the convergence test are presented in Figure 37 and Figure 38.

7.3 Staged and fully coupled solution strategy

In Chapter 4 where the general formulation was addressed two different formulations were presented with respect to formation of the residual (see Section 4.4). Once the formulation is obtained one can use different algorithmic approaches for solution of the coupled problem. Two general strategies are used in the solution of coupled problems[1]:

- Staged solution strategy
- Coupled solution strategy.

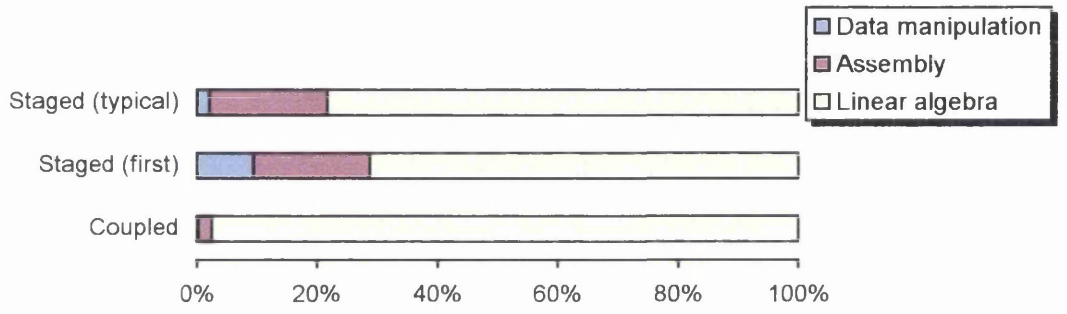
The staged solution strategy treats multiple fields separately. The discrete model of each field can be developed separately and can also be solved using arbitrary methods. The effect of coupling is handled with data transfer between separate models. This solution strategy is quite commonly adopted due to its straightforward implementation procedure if one already has the code for the separate solution of each field. In this case the result of one field represents the input for the other. The modification of the staged strategy is a multi-staggered solution strategy in which partial de-coupling is made of the full system of equations. The full system is partitioned into smaller subsystems based on the assumption that the variables of the other field are temporarily frozen. Each subsystem is then solved separately. In this case the solution of the problem can be performed using the same code for all unknown fields reducing the size of the equation system, which has to be solved at once. The transfer of large amounts of data between separate solvers as in the case of staggered solution procedures is also avoided.
When the coupled solution strategy is used the problem is treated as an indivisible whole. The discrete models are tightly coupled and the system is solved for all unknowns at once.
Within the scope of this work the two solution strategies are compared on an example where a circular rod is heated by a fixed single coil as presented in Figure 35. Two different solution approaches were used:

- Coupled solution strategy
- Multi-staggered solution strategy

The comparison of the solution strategies, was performed using CDriver with the same input files and element routines in both cases. During a single iteration the time measurements were performed for the following characteristic operations:

- Assembly of global quantities
- Linear algebra time
  - Triangular decomposition
  - Back substitution and solution update
- Data manipulation time (freezing DOF, solver profile determination, etc...)

In order to present the timings, first the solution times of the entire global system of equations and of each subsystem during the multi-staggered solution have to be considered. The times required for a single Newton-Raphson iteration are presented in Table 19.
Single iteration solution consisted of a solution of a single global system in the case of coupled and of a single solution of the separate magnetic, thermal and mechanical systems in the case of the multi-staggered solution. It is important to emphasize that the equivalent of a single global coupled iteration consists of several local iterations of each separate field.

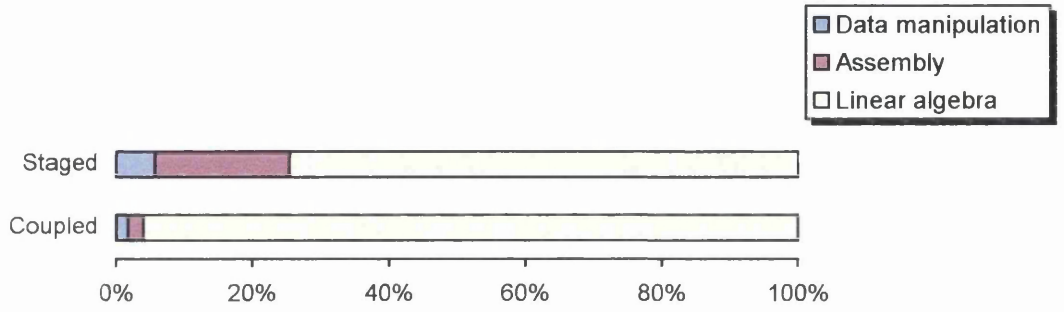


	Coupled	**Stagger first step**	**Stagger typical step**
Iteration time	35.6	8.85	8.07
Number of DOF	21597	12224/ 3145/ 6228	12224/ 3145/ 6228
Average bandwidth	518	298/75/147	298/75/147
Data manipulation time	0.06 s (0.17 %)	0.83 s (9.38 %)	0.16 s (1.98 %)
Assembly time	0.82 s (2.3 %)	1.71 (19.32 %)	1.59 s (19.7 %)
Linear algebra time	34.72s(97.53%)	6.31 s (71.3 %)	6.23 s (78.32 %)

Table 19 Solution times for single iterations

It is obvious that the time required for solution of the global system is significantly higher than solving each of the partial systems. From the data presented in Table 19 it is clear that in the case of the coupled system the majority of time is required for linear algebra operations while in multi-staggered solutions the role of assembly and data handling becomes more important. This is a consequence of the fact that the linear algebra time grows quadratically with the average bandwidth size. If one considers the first iterations then the amount of data handling is higher since the profile has to be determined for each partial system while during subsequent

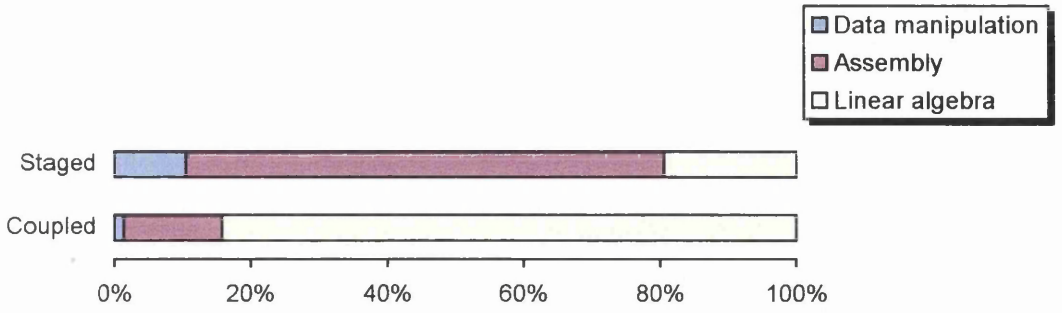
iterations the amount of data handling is reduced significantly. The presented data are important in order to allow a better insight into the following comparisons.
The first comparison, which will be discussed, is the comparison of timing for a single time step in a case when the temperature dependencies of the material parameters are linear. In such a case the nonlinear behaviour of the problem is weakened. The thermal and magnetic subproblems are nonlinear due to the heating source term while the mechanical subproblem is linear. For comparison the same example was used as in the previous case. The coupled analysis requires three global iterations to obtain a convergent solution while staged solution requires six global iterations consisting of 36 local iterations (a single set of global iterations consisted of 2 magnetic, 2 thermal and 2 mechanical iterations). The timings are presented in Table 20.



	Coupled	**Stagger**
Number of DOF	21597	12224/ 3145/ 6228
Num of global iteration	3	6 (36 local iterations)
Iteration time	108.5	102.82
Average bandwidth	518	298/75/147
Data manipulation time	1.8 s (1.66 %)	5.85 s (5.69 %)
Assembly time	2.52 s (2.33 %)	20.32 (19.76 %)
Linear algebra time	104.03 s (96.01 %)	76.65 s (74.55 %)

Table 20 Comparison of a single time step

From the results presented in Table 20 one can conclude that the timing results for both analyses are comparable.
The same comparison was performed on a smaller mesh size and the results are presented in Table 21.



	Coupled	**Stagger**
Number of DOF	5115	2574/861/1680
Num of global iteration	3	6 (39 local iterations)
Iteration time	12.9	30.54
Average bandwidth	344	177/59/115
Data manipulation time	0.17 s (1.32 %)	3.2 s (10.49 %)
Assembly time	1.86 s (14.42 %)	21.42 (70.14 %)
Linear algebra time	10.87 s (84.26 %)	5.92 s (19.38 %)

Table 21 Comparison of a single analysis step on a smaller mesh size

In the case of a smaller mesh size problem the percentage of time spent for linear algebra is reduced significantly since the assembly procedure becomes the most time consuming task.

The performance issues were studied on the heat treatment example with fixed coil in the centre of the rod (see Figure 35). The complete heating time was 24 s and at 12 s the temperature reached the Curie temperature where the magnetic phase transformation occurs instantly dropping the value of magnetic permeability from $90\,\mu_0$ to μ_0. The results are presented in the form of diagrams where the number of global iterations, number of total iterations, computing time and normalized computing time are plotted versus time.

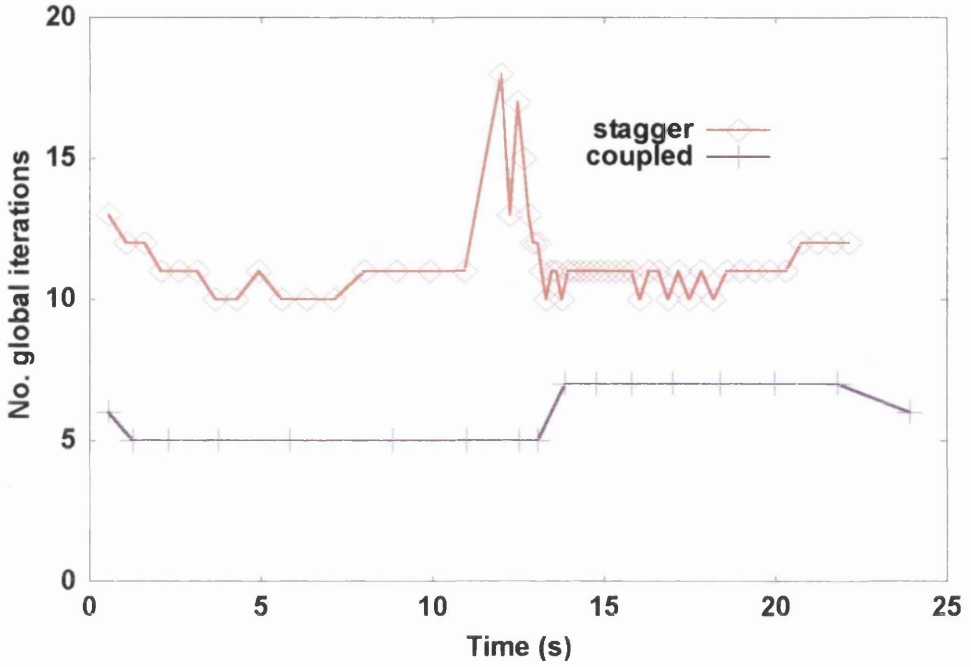


Figure 39 Comparison between solution procedures: number of global iterations vs. time

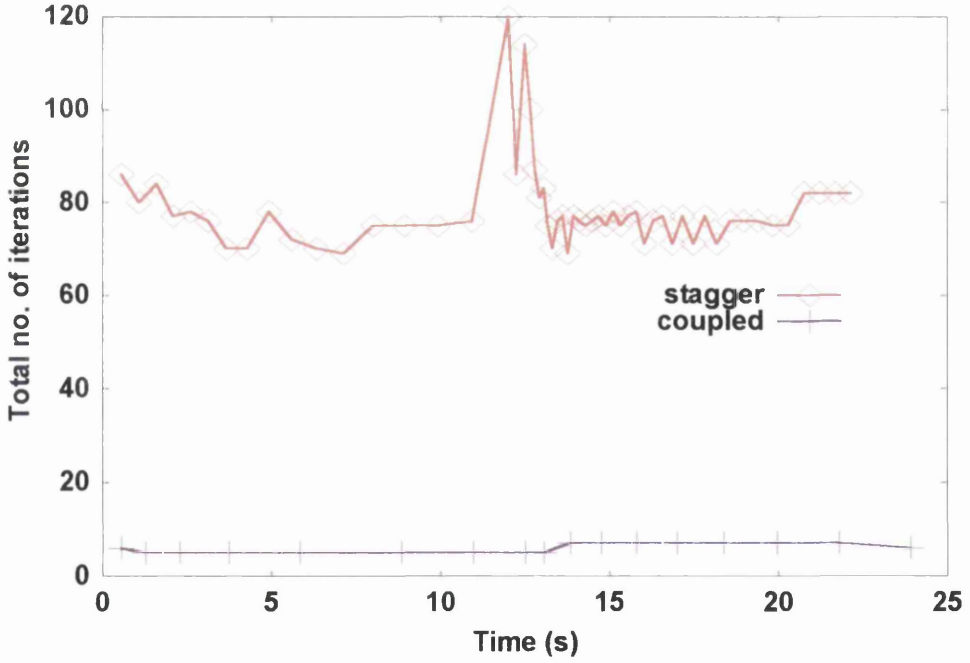


Figure 40 Comparison between solution procedures: total number of iterations vs. time

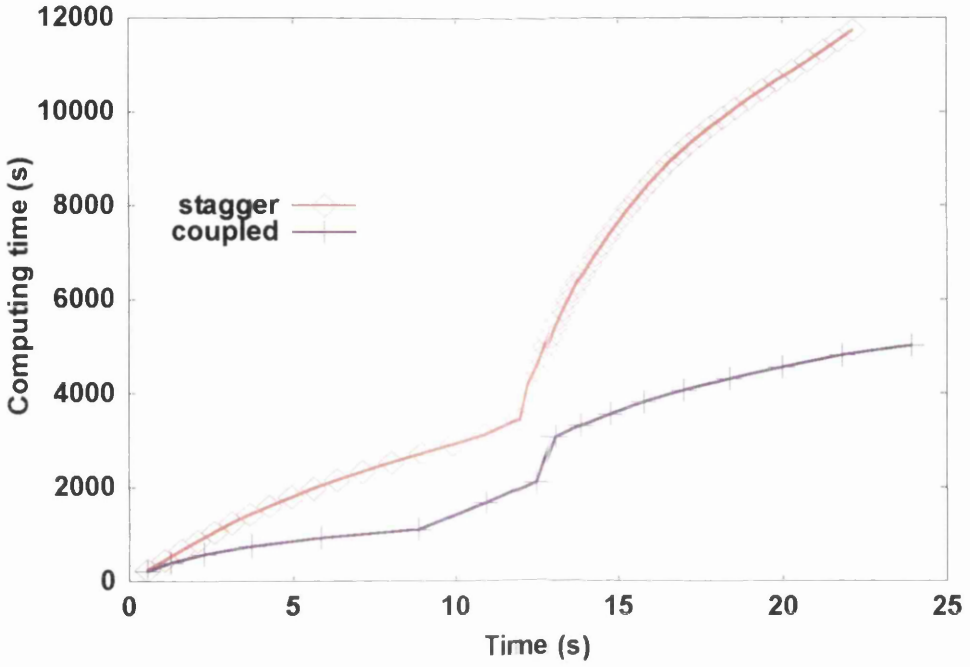


Figure 41 Performance comparison: computing time vs. time

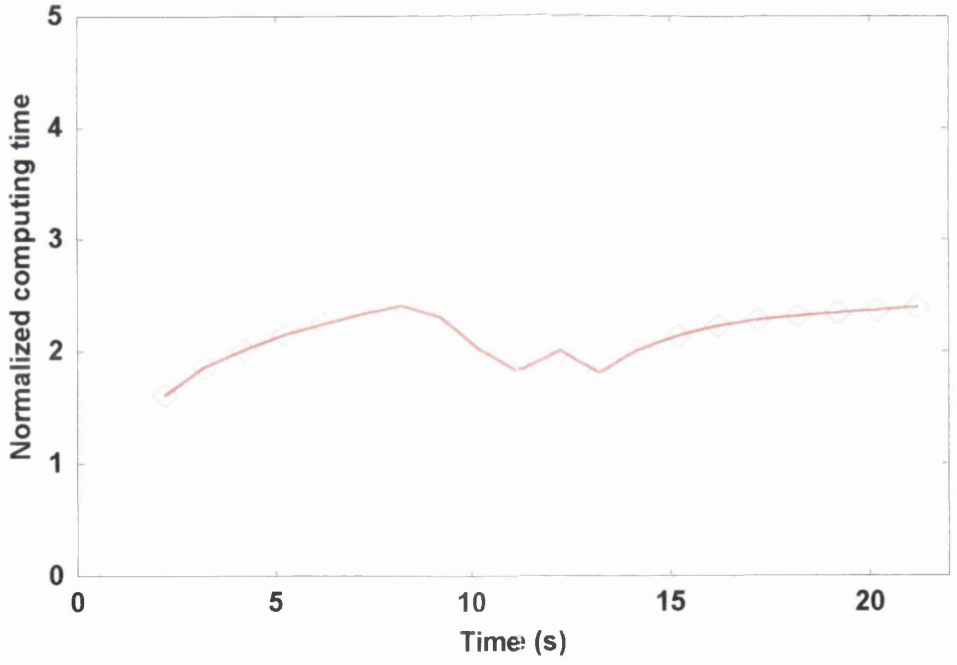


Figure 42 Performance comparison: normalized computing time of staggered solution strategy with respect to coupled strategy vs. time

First, the number of global iterations are compared between both solution procedures. The results are presented in Figure 39 where the number of global iterations is plotted vs. time. Numerical problems were expected at the Curie temperature where the phase transformation occurs. At temperatures above the Curie temperature ferromagnetic material loses its permanent magnetism.
It is clear from the results presented in Figure 39 that the convergence of the coupled numerical procedure is better since it requires fewer time steps and it also passes the Curie temperature without any significant numerical problems. On the other hand during the multi-staggered solution, the time step is significantly reduced when the temperature reaches the Curie temperature.
In Figure 40 the total number of iterations is plotted vs. time for the same example.
Performance issues can be discussed using the diagrams presented in Figure 41 and Figure 42. It is clear from both diagrams that the coupled solution performs better since its rate of convergence is higher. In the region of high material non-linearity the time step reduction significantly affects the performance of the staggered solution procedure while the coupled solution converges without any significant step reductions.

Finally it must be emphasized that the performance issue clearly depends on the size and the complexity of the example and therefore the presented results are illustrative and they are by no means complete. With increased complexity of the problem (material, geometrical non-linearity) the coupled solution procedure should perform better since it has better convergence. But if the problem becomes to large, in terms of degrees of freedom, then the solution of the entire global system can become too expensive and the staggered solution can perform better. Detailed studies of performance issues, which would allow deeper insight into the solution procedure performances, were outside the scope of this work.

References

[1] Ahmed K. Noor, *Computational structures technology: leap frogging into the twenty-first century*, Computers & Structures, Vol 73, p.p. 1-31, 1999.

[2] K.F. Wang, S. Chandrasekar, and Henry T.Y. Yang, *Finite-Element Simulation of Induction Heat Treatment*, Journal of Materials Engineering and Performance, Vol. 1 (1),pp. 97-113, 1992.

[3] C.V. Dodd,W.E. Deeds, *Analytical solution to eddy-current probe coil problems*, Report ORNL-TM-1842, 1967

[4] E.J.W. ter Matten,J.B.M. Melissen, *Simulation of Inductive Heating*, IEEE Trans. Magn., Vol 28(2),pp.1287-1290, 1992

[5] C. Chaboudez, S. Clain, R. Glardon, D. Mari, J. Rappaz and M. Swierkosz, *Numerical modelling in induction heating for Axisymmetric*, IEEE Trans. Magn., Vol 33 (1),pp. 739-745, 1997

[6] T.P. Skoczowski, M.F. Kalus, *The Mathematical Model of Induction Heating of Ferromagnetic pipes*, IEEE Trans. Magn., Vol 25 (2),pp. 1228-1231, 1992

8 INDUCTION HEAT TREATMENT

The implemented magneto-thermal-mechanical model was used for modelling of the heat treatment process. There are not many papers available on numerical solutions of such a problem[1]-[2].
The geometry of the example is presented in Figure 43 where a cylindrical rod is inserted into the centre of a single circular coil that moves sequentially along the rod in the heating stage of the process. After the heating stage is finished the quenching stage begins where the rod is rapidly cooled to room temperature using liquid quenchant.

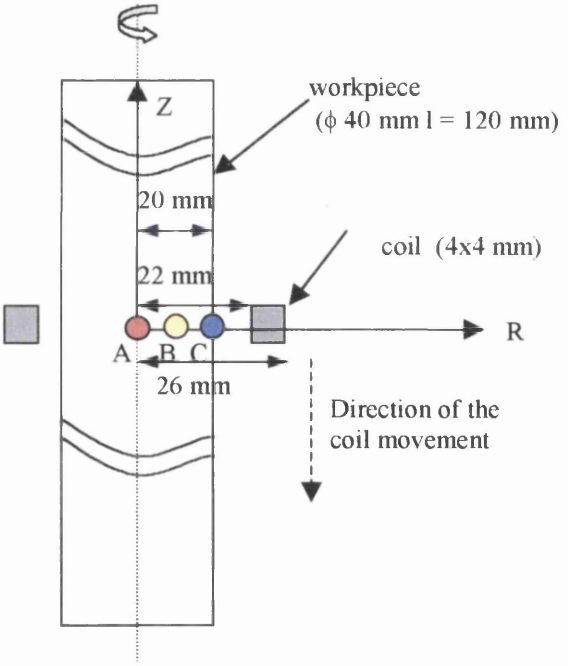


Figure 43 Schematic of the induction heating of a cylindrical rod by a moving single circular coil.

In order to simulate the coil movement the type of elements lying in the coil path were sequentially changed from air to coil according to the current coil position. After the coil movement the elements at the former coil position were changed back

to air while the elements belonging to the new coil position were assigned to coil specification.
The length of the coil path was 80 mm. The sequential movement of the coil consisted of 20 steps. At each step the coil was fixed for 2.5 s and then displaced for 4 mm to a new position.

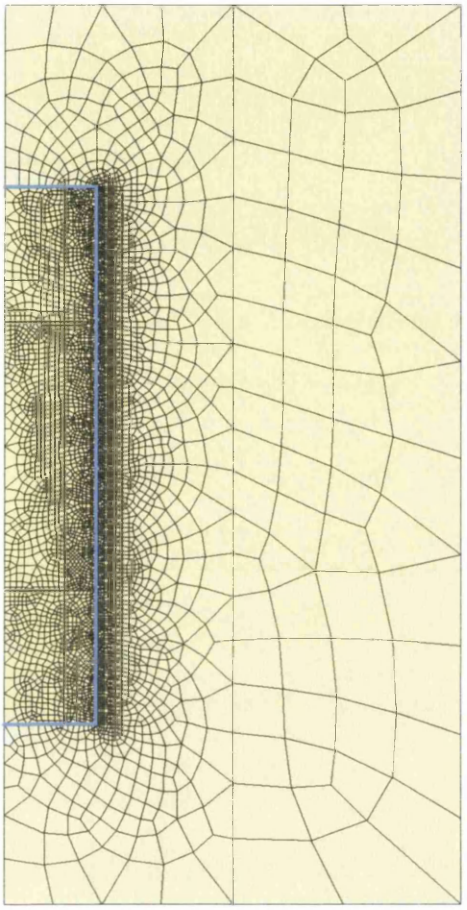


Figure 44 The finite element mesh used for calculation of the heat treatment example

For simulation of the heat treatment process five different elements were used as follows:

1. Air
2. Coil
3. Workpiece
4. Cooling boundary
5. Transition

The transition elements were used to bridge the gap between the regions where elements with different number of degrees of freedom per node were used. In air and coil only the two magnetic and one thermal degrees of freedom were calculated while in the workpiece the displacement field was calculated additionally. The transition element has one (1) or two nodes (2) with five degrees of freedom while the remainder of the nodes have three degrees of freedom as presented in Figure 45.

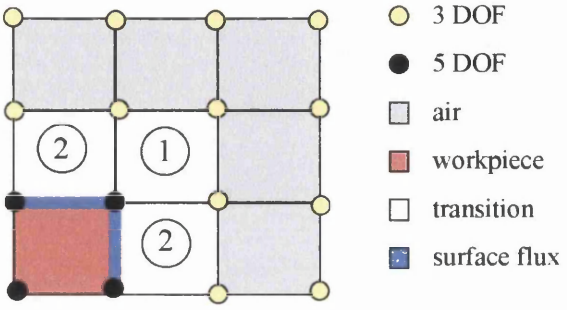


Figure 45 Transition elements

The cooling stage was simulated with thermal surface flux elements with five degrees of freedom per node as illustrated in Figure 45. To simulate the cooling medium the value of the convection coefficient was instantly changed from the value for the air convection to the one of liquid quenchant.

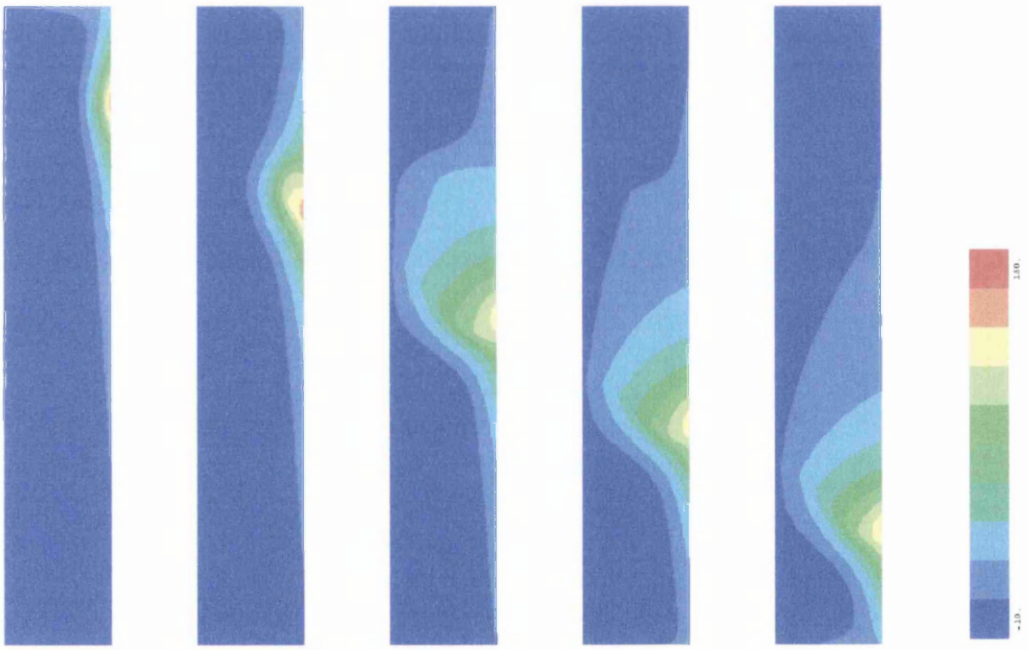


Figure 46 Heat treatment: distribution of A_{im} in the workpiece during heating (time 2.5,15,27.5,40 and 52.5 s)

The magnetic quantities were evaluated during the heating stage. The distribution of the magnetic field inside the workpiece is presented in Figure 46 where the evolution of the imaginary component of the magnetic vector potential is presented during the heating stage. Time evolution of the absolute value of the magnetic vector potential for the points labelled A, B and C in Figure 43 is presented in Figure 47.

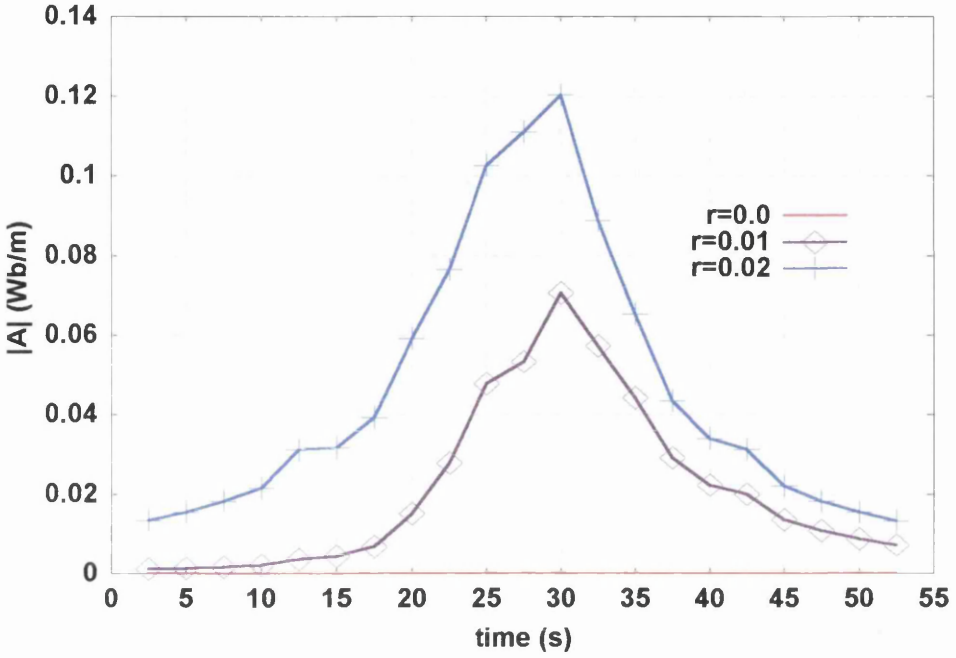


Figure 47 Histories of the absolute value of the magnetic vector potential for the three points in the centre of the rod at z=0 (r=0.02,0.01 and 0)

The evolution of the temperature field is presented in Figure 48 for the heating stage while the temperature distribution during the cooling stage is presented in Figure 49.

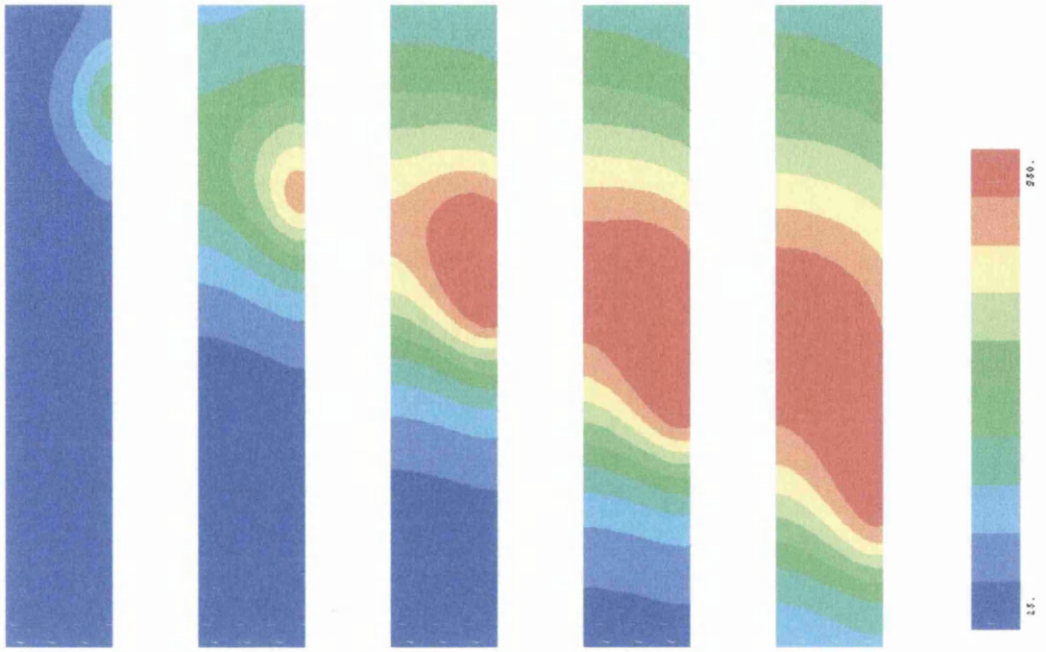


Figure 48 Heat treatment: temperature distribution in the workpiece during heating (time 2.5,15,27.5,40 and 52.5 s)

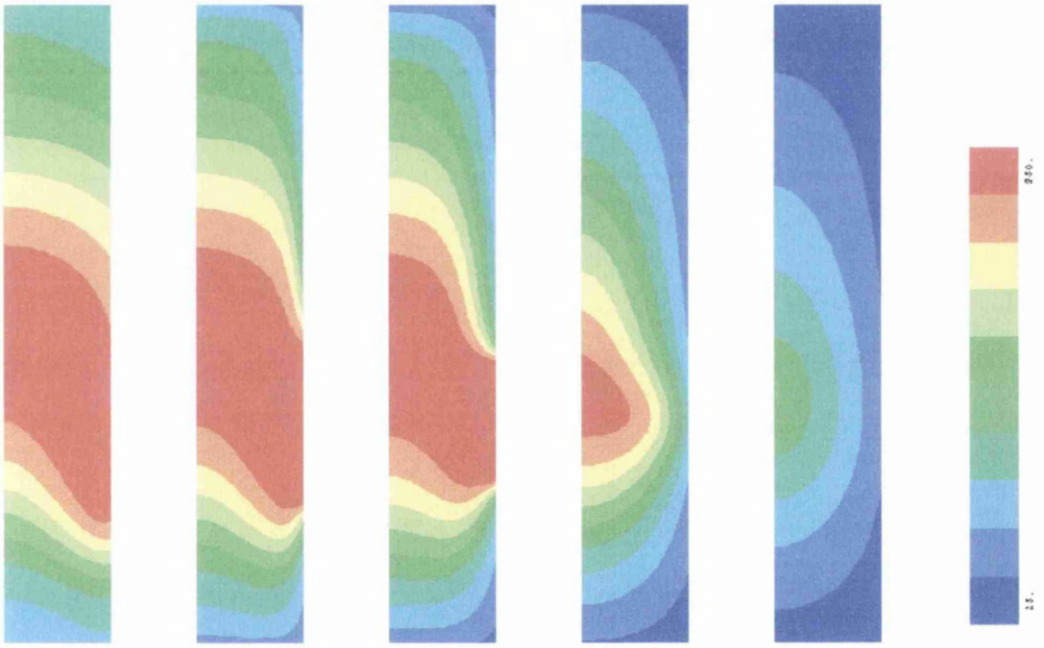


Figure 49 Heat treatment: temperature distribution in the workpiece during cooling (time 52.6, 53.7, 56.1, 60.1, 64.9 and 87.3 s)

The temperature histories for the points in the centre of the rod are presented in Figure 50.

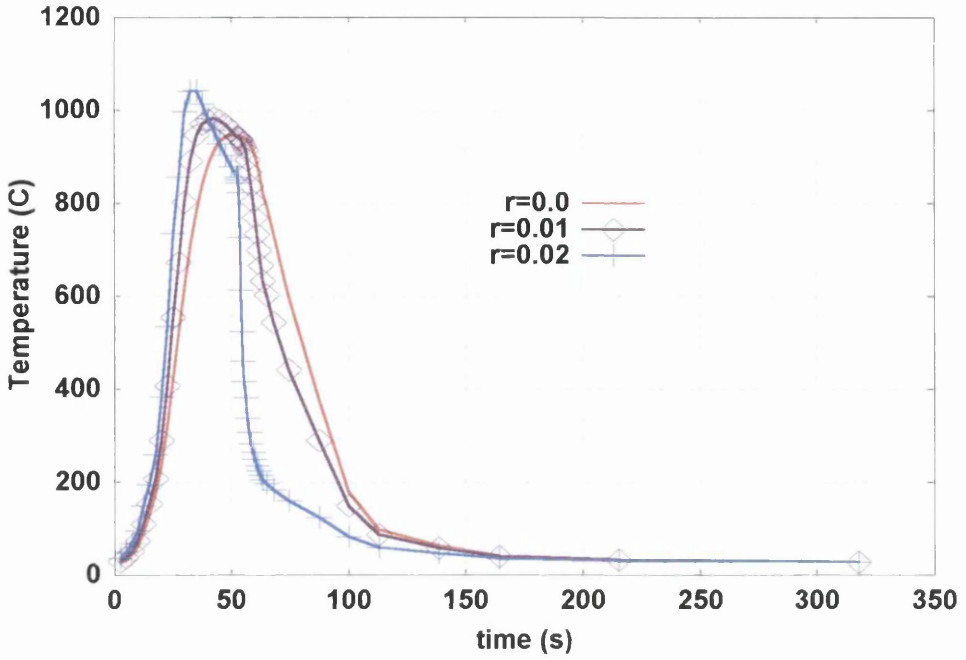


Figure 50 Temperature histories for the three points in the centre of the rod at z=0 (r=0.02,0.01 and 0)

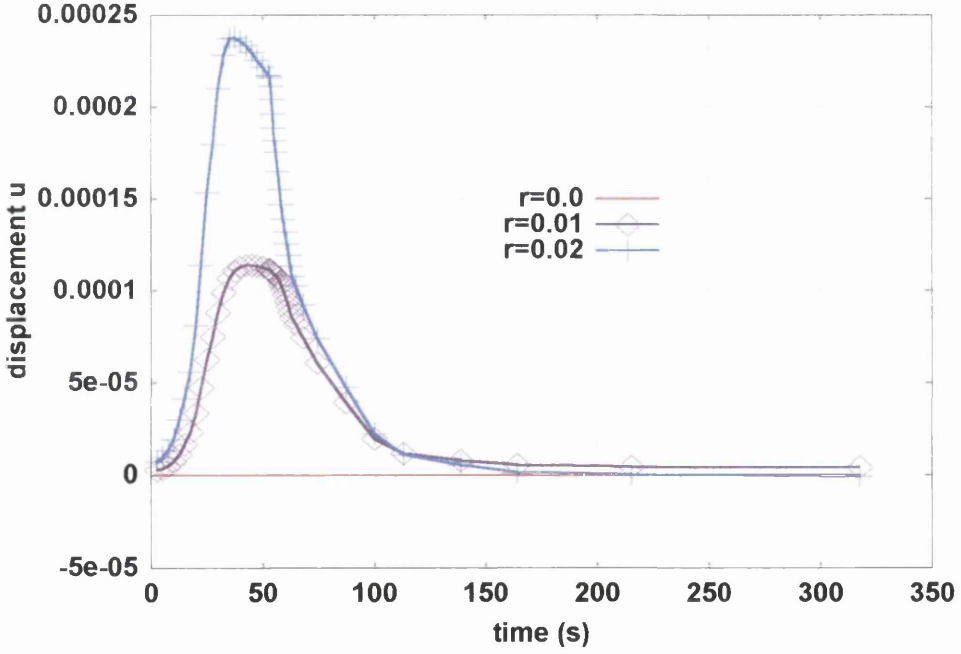


Figure 51 Radial displacement (u) history for the three points in the centre of the rod at z=0 (r=0.02,0.01 and 0)

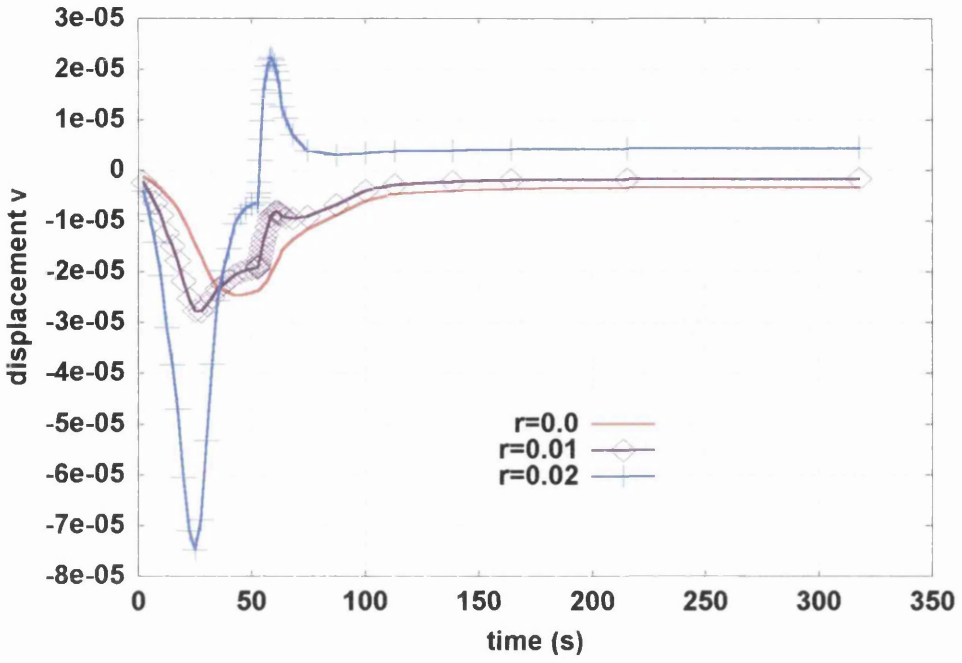


Figure 52 Axial displacement (v) history for the three points in the centre of the coil at z=0 (r=0.02,0.01 and 0)

The displacement histories for selected profiles are presented in Figure 51 and Figure 52 for both radial and axial displacement.

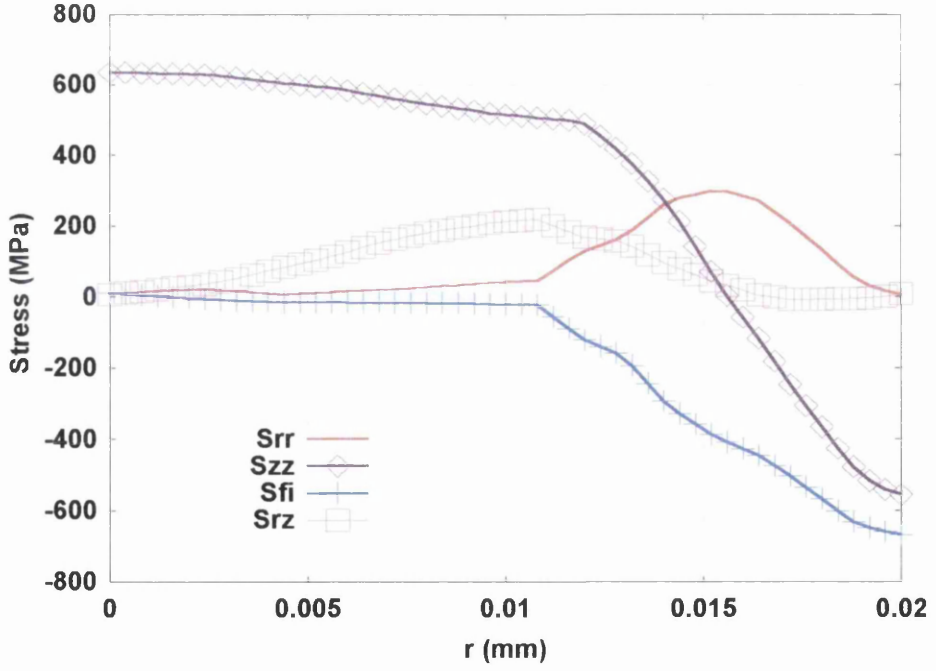


Figure 53 Distribution of stresses after the heat treatment process at z=0

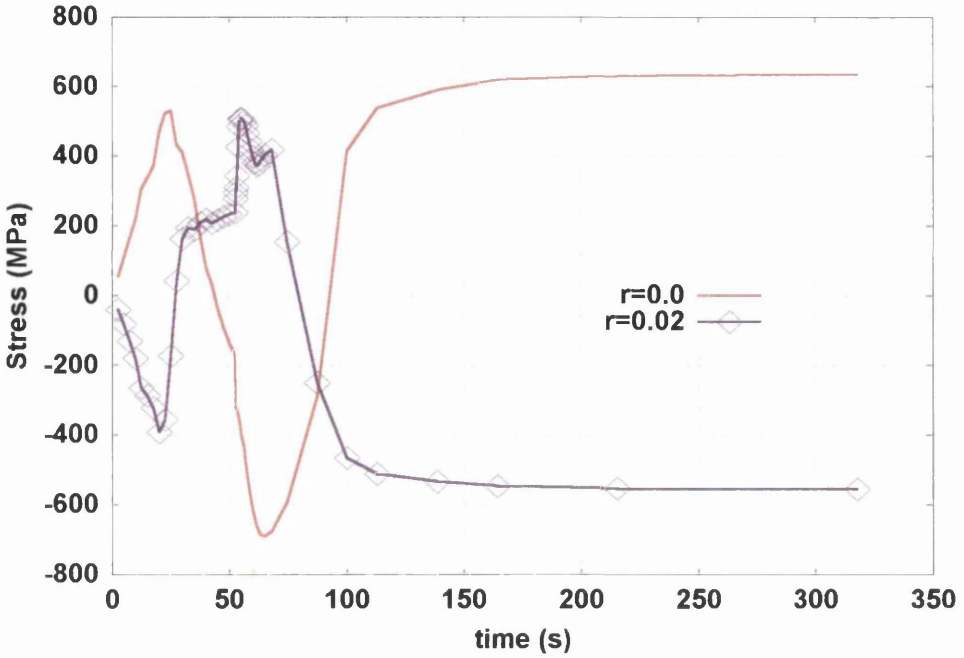


Figure 54 Histories of the axial stresses σ_{zz} at the points A and C

The stress distribution in the workpiece profile (z=0, r =0.0 ... 0.02) after the heat treatment process is illustrated in Figure 53 while the histories of the axial stresses are presented in Figure 54. Both the hoop ($\sigma_{\phi\phi}$) and axial stresses (σ_{zz}) are compressive to a depth of approximately 5 mm from the surface and therefore the objective of the heat treatment process is satisfied.

References

[1] K.F. Wang, S. Chandrasekar, and Henry T.Y. Yang, *Finite-Element Simulation of Induction Heat Treatment*, Journal of Materials Engineering and Performance, Vol. 1 (1),pp. 97-113, 1992.

[2] K.F. Wang, S. Chandrasekar, and Henry T.Y. Yang, *An Efficient 2D Finite Element Procedure for Quenching Analysis with Phase Change*, ASME J. Eng.Ind., 1992.

9 CONCLUSIONS

In this work a computational environment for the solution of coupled non-linear problems has been presented. First an overview of solution environments and relevant scientific computing techniques was provided in the second chapter.
In the third chapter the co-operative approach to the solution of non-linear coupled problems was introduced describing the *AceGen* symbolic code generator *Computational Templates* package. As an important part of the system a finite element driver concept was presented in detail since *CDriver* has been developed within the scope of this work.
In the fourth chapter the general formulation for implicit solution of transient coupled nonlinear systems has been stated which was implemented in subsequent chapters. The purpose of this chapter was also to introduce the notation used throughout the work. Chapter five introduced the finite element method and its implementation aspects.
Using the co-operative approach, an implicit finite element model of inductive heat treatment has been implemented where the magnetic, thermal and displacement fields are coupled. First the theoretical aspect of the individual fields were addressed and the corresponding numerical model were developed and verified in the sixth chapter. Based on models of individual fields, coupled magneto-thermal and magneto-thermal-mechanical model were derived and verified in the seventh chapter. As a part of the verification process the aspects of different solution strategies for coupled problems were also investigated for this example. In the last chapter the example of inductive heating heat treatment has been solved and the results obtained were presented.

It has been shown that using the presented solution approach complex problems can be solved in a reasonable amount of time significantly reducing the amount of work related to low-level programming and testing. Using such a co-operative approach more time can be dedicated to the physical aspects of the problem and less to its implementation details.

In the future several improvements of the *CDriver* are required in order to solve large-scale problems including high performance sparse solver, remeshing and domain decomposition algorithms. The developed magneto-thermo-mechanical model should be improved to cover large strain plasticity, which is necessary for simulation of forming technologies such as die-less forming.

A. APPENDIX

Input for comparison of solution strategies

Analysis - coupled solution strategy

```
FMTDTemperatureList = {}; FMTDIterationList = {}; FMTDTimeList = {};
CurTime = 0.;
PrevTime = 0.;
```

```
step = EndTime / NStep
```

```
SMTNextStep[step, 1];
```

```
CurTime0 = SMTSessionTime[];
While[(SMTRData["Time"] - step) < EndTime && step > 0.01,
 While[SMTNewtonIteration[] > 10^-10 && SMTIData["Iteration"] < 10 ,
  SMTStatusReport[];]; SMTStatusReport[];
 If[SMTIData["Iteration"] < 10
  , PrevTime = CurTime;
  CurTime = SMTSessionTime[];
  AppendTo[FMTDTimeList, {SMTRData["Time"], CurTime - CurTime0}];
  AppendTo[FMTDIterationList, {SMTRData["Time"], SMTIData["Iteration"]}];
  If[SMTIData["Iteration"] < 6, step = 2 step;];
  SMTNextStep[step, 0];
  Print["Next ", step];
  , SMTStepBack[]; step = step / 2.; SMTNextStep[step, 0];
  Print["Back ", step];
   ];
]
```

Analysis - staggered solution strategy

■ Definitons of the freezing functions for solutions of separate fields

```
alldof = SMTNodeData["DOF"];
```

```
ϕDfreeze =
  Map[
   If[Length[#] == 5, {#[[1]], #[[2]], If[#[[3]] > -1, -2, -1],
      If[#[[4]] > -1, -2, -1], If[#[[5]] > -1, -2, -1]}, {#[[1]], #[[2]]}] &,
   alldof];
```

```
ADfreeze =
  Map[
   If[Length[#] == 5, {If[#[[1]] > -1, -2, -1], If[#[[2]] > -1, -2, -1],
      #[[3]], If[#[[4]] > -1, -2, -1], If[#[[5]] > -1, -2, -1]},
     {If[#[[1]] > -1, -2, -1], If[#[[2]] > -1, -2, -1]}] &, alldof];
```

```
Aϕfreeze =
  Map[
   If[Length[#] == 5, {If[#[[1]] > -1, -2, -1], If[#[[2]] > -1, -2, -1],
      If[#[[3]] > -1, -2, -1], #[[4]], #[[5]]},
     {If[#[[1]] > -1, -2, -1], If[#[[2]] > -1, -2, -1]}] &, alldof];
```

```
TemperatureList = {}; IterationList = {}; TimeList = {};
GlobIterationList = {};
```

```
i = 0; geniter = 0; PrevTime = 0.; CurTime = 0.;
```

```
step = EndTime / NStep
```

```
SMTNextStep[step, 1];
```

```
CurTime0=SMTSessionTime[];
While[(SMTRData["Time"]-step) <EndTime && step>0.01,
  err=10^10;i=i+1;
  Print["STEP ",i," of ",NStep];ndiver=True;ngstep=0;
  While[ndiver && err>10^-10 && SMTIData["Iteration"]<200,
    ngstep++;
    Print["ERR ",err];
    SMTNodeData["DOF",ϕDfreeze];SMTSetSolver[2];
    err=SMTNewtonIteration[];SMTStatusReport["mag"];
    IterCounter=SMTIData["Iteration"];
    While[SMTNewtonIteration[]> 10^-11 && (ndiver=SMTIData["Iteration"]-
IterCounter<10) ,SMTStatusReport["mag"];];
    SMTStatusReport["mag"];
    If[ndiver,
      SMTNodeData["DOF",ADfreeze];SMTSetSolver[2];
      err+=SMTNewtonIteration[];SMTStatusReport["therm"];
      IterCounter=SMTIData["Iteration"];
      While[SMTNewtonIteration[]> 10^-11  && (ndiver=SMTIData["Iteration"]-
IterCounter<10) ,SMTStatusReport["therm"];];
      SMTStatusReport["therm"];
      ];
    If[ndiver,
      SMTNodeData["DOF",Aϕfreeze];SMTSetSolver[2];
      err+=SMTNewtonIteration[];SMTStatusReport["mech"];
      IterCounter=SMTIData["Iteration"];
      While[SMTNewtonIteration[]> 10^-11  && (ndiver=SMTIData["Iteration"]-
IterCounter<10) ,SMTStatusReport["mech"];];
      SMTStatusReport["mech"];
      ];
    ];
  If[ndiver && SMTIData["Iteration"]<150
    ,PrevTime=CurTime;
    CurTime=SMTSessionTime[];
    AppendTo[TimeList,{SMTRData["Time"],CurTime-CurTime0}];
    AppendTo[IterationList,{SMTRData["Time"],SMTIData["Iteration"]}];
    AppendTo[GlobIterationList,{SMTRData["Time"],ngstep}];
    If[ngstep<20,step=2 step;];
    SMTNextStep[step,0];
    Print["Next ",step,"  ",ngstep];
    ,SMTStepBack[];step=step/2;SMTNextStep[step,0];Print["Back ",step];
    ];
  ]
```

Input for analysis of inductive heat treatment example

Step 1: Mesh input reading

```
SMTStructure["Input" → "small_induction_bound_centre_supp.inp"];
```

Step 2: Setting the initial temperature

```
nodes =
  Union@@
   (SMTElementData[#, "Nodes"] & /@
     Flatten[Position[SMTElementData["SpecIndex"], 2]]);
```

```
SMTNodeData[#, "at", {0., 0., 25., 0., 0.}] & /@ nodes;
SMTNodeData[#, "ap", {0., 0., 25., 0., 0.}] & /@ nodes;
```

Step 3: Coil path determination

```
coilelements = Position[#[[2]] & /@ SMTElements, 1] // Flatten;
```

```
ylist = SMTNodes[[#]][[3]] & /@ SMTElements[[#]][[3]] & /@ coilelements;
```

```
coilrange = Table[0.06 - (i 0.004), {i, 0, 30}];
```

```
rr = {}
For[i = 1, i ≤ (Length[coilrange] - 1), i++,
 a = coilrange[[i]];
 b = coilrange[[i + 1]];
 AppendTo[rr, {}];
 For[k = 1, k ≤ Length[coilelements], k++,
  ylist = SMTNodes[[#]][[3]] & /@ SMTElements[[coilelements[[k]]]][[3]];
  For[j = 1, j ≤ 4, j++,
   If[ylist[[j]] < a && ylist[[j]] > b,
    AppendTo[rr[[i]], coilelements[[k]]]]]
 ]
]
```

```
CoilPath = Intersection[#, coilelements] & /@ rr;
```

```
SMTElementData[#, "SpecIndex", 7] & /@ coilelements;
```

Step 4: Analysis

```
PositionTime = 2.5; PStep = 5.;
```

```
FMTDTemperatureList = {}; FMTDIterationList = {}; FMTDTimeList = {};
CurTime = 0.; PrevTime = 0.;
LabelList = {};
```

```
step = PositionTime / PStep
```

```
InitialStep = step; load = 1;
```

```
CurTime0 = SMTSessionTime[];
For[i = 5, i < 26, i++,
  Ptime = 0.0; step = InitialStep;
  SMTElementData[#, "SpecIndex", 1] & /@ CoilPath[[i]];
  EndTime = SMTRData["Time"] + PositionTime; Print["EndTime :", EndTime];
  While[(SMTRData["Time"]) < EndTime && step > 0.01,
   If[(EndTime - (SMTRData["Time"] + ( step))) < 0,
    step = EndTime - SMTRData["Time"];];
   SMTNextStep[step, load]; load = 0.; Print["Heating Next ", step];
   While[ConvRet = SMTConvergence[10^-10, 10, "Ignore"] ,
    SMTNewtonIteration[]; SMTStatusReport[];];
   If[ConvRet === Indeterminate,
    (* divergence *)
    SMTStepBack[]; step = step / 2.;
    Print["Back ", step];,
    (* convergence *)
    PrevTime = CurTime;
    CurTime = SMTSessionTime[];
    AppendTo[FMTDTimeList, {SMTRData["Time"], CurTime - CurTime0}];
    AppendTo[FMTDIterationList, {SMTRData["Time"], SMTIData["Iteration"]}];
    If[SMTIData["Iteration"] < 6,
     If[(EndTime - (SMTRData["Time"] + (2 step))) < 0,
       step = EndTime - SMTRData["Time"];, step = 2 step;];
    ];
   ];
  ];
  Print["Coil move ", i, "/", Length[CoilPath]];
 ];
```

Step 5: Analysis of the cooling stage

```
(* setup of the cooling boundary *)
BoundEl = Position[#[[2]] & /@ SMTElements, 6] // Flatten;
SMTElementData[#, "SpecIndex", 8] & /@ BoundEl;
```

```
(* turning the coil off *)
SMTESpecData[1, "Data", {1.2566371 10^-6, 60., 0., 5.75 10^7}]
```

```
EndTime = SMTRData["Time"] + 120.
```

```
step = 0.1;
```

```
While[(SMTRData["Time"] - step) < EndTime && step > 0.005,
  While[ConvRet = SMTConvergence[10^-10, 10, "Ignore"] ,
   SMTNewtonIteration[]; SMTStatusReport[];];
  If[ConvRet === Indeterminate,
   (* divergence *)
   SMTStepBack[]; step = step / 2.; SMTNextStep[step, 0];
   Print["Back ", step];,
   (* convergence *)
   PrevTime = CurTime;
   CurTime = SMTSessionTime[];
   AppendTo[FMTDTimeList, {SMTRData["Time"], CurTime - CurTime0}];
   AppendTo[FMTDIterationList, {SMTRData["Time"], SMTIData["Iteration"]}];
   If[SMTIData["Iteration"] < 6, step = 2 step;];
   SMTNextStep[step, 0];
   Print["Next ", step];
    ];
 ];
```